Predicting Protein Structure Using Parallel Genetic Algorithms

THESIS

George H. Gates, Jr.
Captain, United States Air Force

AFIT/GCS/ENG/94D-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-03

Predicting Protein Structure Using Parallel Genetic Algorithms

THESIS

George H. Gates, Jr.
Captain, United States Air Force

AFIT/GCS/ENG/94D-03

AFIT/GCS/ENG/94D-03

Predicting Protein Structure
Using Parallel Genetic Algorithms

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

George H. Gates, Jr., B.S.

Captain, United States Air Force

13 December 1994

Table of Contents

List of Figures

List of Tables

Abstract

The protein folding problem is a Grand Challenge problem in biochemistry. The challenge is to reliably predict the natural three-dimensional structure of a polypeptide given only the arrangement of its constituent atoms. Energy minimization is a classical method used to predict the natural conformation of such molecules. This study enhances an energy model implementation that is integrated with genetic algorithms to minimize polypeptide energy and predict natural structure.

Genetic algorithms (GAs) are robust, semi-optimal search techniques modeled after evolution theories. The most commonly used simple genetic algorithms (SGAs) consist of three genetic operators: crossover, mutation, and selection. The operators manipulate populations of strings that represent solutions to specific domain problems. Deceptive problems limit the effectiveness of SGAs and their convergence is extremely dependent on several GA parameters. Fast messy GAs (fmGAs) are variants of messy GAS that reduce the exponential time complexity to polynomial. All of these genetic algorithm variations can be parallelized with several parallel communication strategies influencing their computational performance. This investigation evaluates the merits of parallel SGAs and fmGAs for minimizing the potential energy of a pentapeptide, [Met]-enkephalin.

AFIT's energy model is compared to a similar model in a commercial package called QUANTA. Differences between the two models are identified and resolved to enhance GAs' abilities to correctly fold molecules. The steps required to unify the behavior of the two implementations is presented.

The effectiveness of SGAs while minimizing the potential energy of [Met]-enkephalin is shown to be highly dependent on the choice of population size and mutation rate. It is also demonstrated that choosing parameters from the guidelines proposed by Schaffer's work cause SGAs to realize their near-optimal performance on this particular real-world application.

Parallel SGAs, using appropriate parameters, are capable of finding near-optimal conformations of [Met]-enkephalin. Parallel fmGAS should ultimately be able to find better solutions in less time. The experiments performed in this investigation are designed to determine the limitations of parallel SGAs and fmGAs applied to polypeptide energy minimization. The results from these experiments are used to identify research directions critical to increasing the maturity of fmGAs.

Predicting Protein Structure
Using Parallel Genetic Algorithms

I. Introduction

Since the beginning of the Information Age, computers have been applied to problems that need to be solved quickly and accurately. In the past, when these problems couldn't be solved fast enough by the current technology, new hardware devices were developed that could meet the processing requirements. However, with the impending introduction of TeraFLOP computers[1], we are currently approaching the physical limits of electronic computing devices (17:1–2).

Recent research efforts have turned toward creating general search/optimization algorithms designed to attack these *difficult* problems (34:2–6)(26). There are two areas we are pursuing at the Air Force Institute of Technology (AFIT); parallel computing and semi-optimal algorithms. Parallel computing is a field that tries to describe how problem solutions should be decomposed so computations can be accomplished on many processors simultaneously (54:2–3). Semi-optimal algorithms are being explored as a means to obtain *good* solutions to intractable problems (20, 62). We combine these disciplines using parallel genetic algorithms to solve a variety of complex problems (65, 3, 72, 61).

One problem that AFIT is particularly interested in is the *protein folding problem.* The protein folding problem is an intractable problem contained in a class known as Grand Challenge problems (9). The challenge is to find a method of predicting the 3-dimensional structure of a protein given its defining sequence of amino-acids. An enumerative search of the entire solution space for even the smallest proteins would consume more time than the estimated age of the universe on today's supercomputers!

1.1 Algorithmic Complexity

For large problems, the efficiency of an algorithm depends more on its algorithmic complexity than on the speed of the particular machine on which it's executed. Optimization problems involve

[1] A computer that can process one trillion floating point operations per second

an explicit or implicit search of the entire solution space to guarantee the best solution is found (73:7–8). However, the execution time for such a search typically grows exponentially with respect to the size of the given problem. This growth severely limits our ability to solve practical problems of any significant size. Table 1.1 compares the sizes of problems that could be solved by using either faster computers or better algorithms. The table illustrates why solutions with polynomial time complexity are preferred over exponential solutions.

Table 1.1 Size of Problems Solvable in One Hour (assuming one evaluation per clock cycle)

Hardware Capability	Algorithmic Complexity	Problem Size Solved (n)
GigaFLOP	Exponential (2^n)	41
TeraFLOP	Exponential (2^n)	51
GigaFLOP	Polynomial (n^2)	1,897,366
TeraFLOP	Polynomial (n^2)	60,000,000

As we approach the physical limits of single processor computers, our attention has turned to parallel computer architectures for increased performance. Many architectures exist which exploit different parallelization schemes. The two primary architectures are *single instruction stream, multiple data stream* (SIMD) and *multiple instruction stream, multiple data stream* (MIMD) (54:16–17). The *single program, multiple data* (SPMD) parallel programming model is often implemented on a MIMD architecture (54:528). Another class of parallel computer architecture involves *distributed* computing on existing networks of workstations. Each of these architectures has benefits and limitations with respect to particular applications, communication and dependencies between tasks, and data and task distributions.

Contemporary parallel computers have from two to one thousand processors, and plans are being drawn for architectures with more than one million nodes (83). The major stumbling block in applying parallel computing has been our inability to conceptualize parallel approaches to problem solving. We tend to think and solve problems sequentially, but the sequential solutions to problems rarely transform into quality parallel solutions (54). A few algorithms can be easily parallelized using simple data or control decomposition techniques (60). However, overhead costs, in the forms of task scheduling, task management, synchronization, and load balancing, limit the performance of those programs. Existing parallel program design languages include UNITY (7), Communicating

Sequential Processes (CSP) (47), and Petri-Nets (74). Each method proposes a unique way to treat parallelism as an additional level of abstraction in the program design process.

Parallel performance is typically measured by execution time compared to sequential algorithms. Parallel solutions are said to be *scalable* if additional processors can be used efficiently, and thus reduce the overall execution time (54:6). The goal of parallel algorithm design is to either reduce the time complexity of an algorithm or create an algorithm that is (infinitely) scalable.

1.2 Genetic Algorithms

Genetic algorithms (GAs) are a relatively new class of semi-optimal search/optimization algorithms that exhibit the desired traits listed previously (48)(34:1–2). The execution time of a genetic algorithm is typically dominated by the calculation of a fitness function. This function is problem dependent, but is usually of polynomial time complexity.

Genetic algorithms work on *populations* of solutions called *strings*. Simple GAs perform three basic operations on strings in the population: *selection*, *crossover*, and *mutation* (34). The algorithm steps through these three operations repeatedly until some stopping criteria are met. The execution of a single GA iteration is called a *generation*. Because GAs are loosely based on natural evolution, many of the terms associated with natural evolution are used interchangeably with terms created specifically for genetic algorithms (55).

GAs are easily parallelized because the simplest approach is a SPMD model which puts multiple copies of the same program on each processor and selects the best solution after all processors have finished.

1.3 The Protein Folding Problem

Polypeptides (proteins) are *defined* by their sequence of amino-acids, or *primary structure* (6). The *function(s)* of a protein is determined by its 3-dimensional shape, or *tertiary structure* (58). Current experimental techniques are an effective means to decode the primary structure of a protein. However, the cost associated with the experimental determination of an arbitrary protein's tertiary structure is currently prohibitive. Without a breakthrough in physical experimental techniques or

some other currently unknown field, reliable tertiary structure *prediction* remains as the only viable protein structure identification alternative.

Nuclear magnetic resonance (NMR) spectography and X-ray crystallography are the two experimental techniques for determining the 3-dimensional structure of a protein. However, both approaches can expend more than two years of laboratory effort to find the tertiary structure of a single protein (58). To reduce the gap between the number of known protein sequences and tertiary structures (the difference is now two orders of magnitude and growing (58)), we need to be able to reliably predict the tertiary structure of proteins in a reasonable amount of time.

There are two classical techniques for predicting the tertiary structure of proteins: energy minimization and molecular dynamics. Energy minimization methods use a potential energy function to evaluate the stability of a protein conformation (57). Applied to problems with n atoms, the time complexity of a force-field energy model is $\mathcal{O}(n^2)$, while an exact model is $\mathcal{O}(n^5)$ (58). Since $n > 1000$ is the norm, energy minimization techniques typically use the less computationally intensive force-field models. The topology of the energy landscape contains many local minima which, combined with the huge size of the conformational search space, precludes the use of classical optimization techniques. The probabilistic nature of genetic algorithms can be exploited to escape local minima in an effort to find the global optimum.

Molecular dynamics attempts to simulate the protein folding process. However, the time steps required for this simulation are on the order of one femtosecond (10^{-15} sec) to accurately account for thermal oscillations of the protein. Using current supercomputer technology, only a few hundred picoseconds (10^{-12} sec) of the folding process can be simulated, while the actual folding process may span more than half a second. In addition, the forces used for the simulation are typically calculated using the force fields described above (58:5–7).

1.4 Problem Statement

AFIT's Genetic Algorithm Toolkit contains general implementations of several serial and parallel genetic algorithms (20, 62) and evaluation functions for several domain specific problems (3, 72, 61) (See Figure 1.1). The goal of this investigation is to evaluate the performance of these

GAs as they're applied to the protein folding problem. Specifically, we use genetic algorithms to search for low-energy conformations of [Met]-enkephalin and try to answer the following questions:

1. Which genetic algorithm finds the best solutions and why?
2. What are the performance issues and tradeoffs associated with parallel GAs?
3. How do the parameter settings for GAs affect their performance?
4. Are these techniques generally applicable to the protein folding problem?

1.5 Scope of Investigation and Rationale

There are three major objectives of this investigation. First, we must validate the energy function used to search for minimum energy protein conformations. Validation is in the form of comparison with a proprietary implementation of the same energy model. Then, we want to establish sets of parameters that cause a simple genetic algorithm to produce low-energy conformations of the polypeptide [Met]-enkephalin. Finally, we'll compare the performance of two different genetic algorithms and their parallel implementations. The attainment of these objectives will provide a basis on which to recommend future research directions.

Two important assumptions also limit the scope of this research:

- The GAs in AFIT's Genetic Algorithm Toolkit work correctly.
- The CHARMm energy model implemented by QUANTA is correct.

Additionally, it's assumed that a reader is familiar with basic concepts associated with computer science, discrete mathematics, and statistics as an aid to understanding Chapters IV through VI.

In terms of the protein folding problem, the goal of this research is to determine if genetic algorithms can effectively predict the tertiary structure of proteins within an acceptable amount of time, using serial or parallel computation. However, we do not expect GAs using energy minimization to be capable of folding arbitrarily large proteins (> 200 residues). The intended uses of energy minimization GAs include: determining the tertiary structure of small proteins (< 100 amino acids) and non-linear optical (NLO) polypeptides; folding the interesting portions of large proteins that have not been resolved experimentally (loop regions for example); and identifying promising, locally optimal structures as input for other optimization techniques.

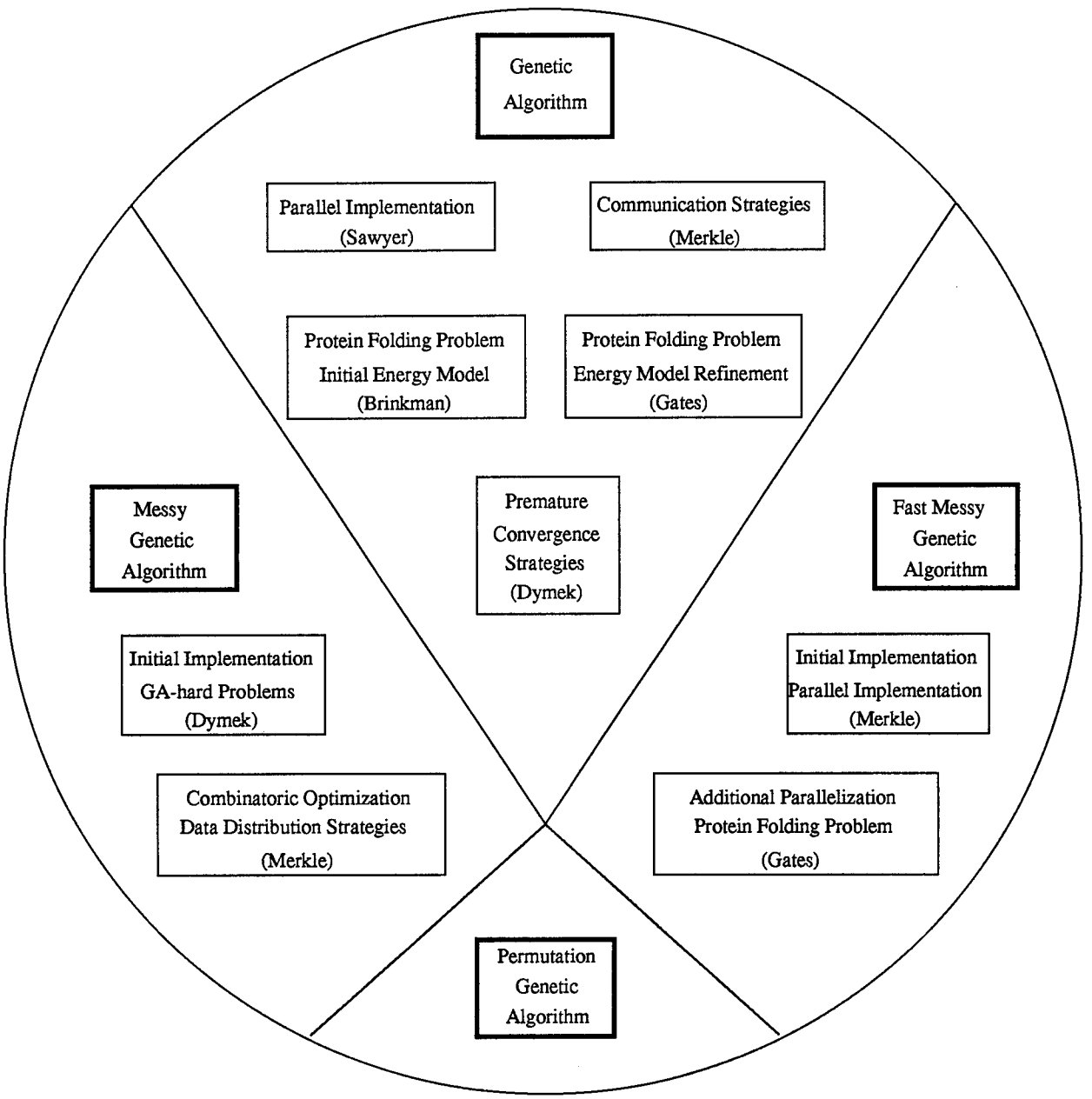


Figure 1.1 AFIT's Genetic Algorithm Toolkit (Current Status)

Solving the protein folding problem implies the ability to reliably predict the tertiary structure of any protein once given the primary structure of that protein. Knowing the function of the various proteins present in our own bodies could lead to many new medical and scientific breakthroughs:

- Preventing or curing disease
- Repairing genetic disorders or birth defects
- Developing disease or pest resistant strains of plants

The solution of the protein folding problem is also significant because it could provide insight into its complementary problem, which is: given a particular function we desire a protein to perform, what is the primary sequence of a protein that will fold into a tertiary structure that will perform that function? The solution of this complementary problem would allow biochemists to design new polypeptides with a single, specific purpose. It would then be technically possible to develop new drugs with little or no side effects.

The application of genetic algorithms to the protein folding problem also provides us with empirical evidence we can use to evaluate their applicability to other practical problems. We are particularly interested in establishing the factors that enhance or limit a GA's capability to perform a robust search and provide near-optimal solutions for a wide variety of problems. Thus, any theoretical or practical knowledge gained from this research could be beneficial to many other domains that contain a search or optimization problem as a major component.

1.6 Methodology

The original energy model implemented at AFIT (3) is enhanced to account for the relationships between an independently variable dihedral angle and all other dihedral angles defined by the same first three atoms. This new structural model is integrated with a simple genetic algorithm. Parametric tests are executed to establish a "good" parameter set for the simple genetic algorithm search process. The energy model is then integrated with a parallel simple genetic algorithm, a fast messy genetic algorithm, and a parallel fast messy genetic algorithm. The performance of these four probabilistic search algorithms is then compared using the enhanced energy model and parameters from the empirically determined best parameter set.

1.7 Summary

Performance gains from computational hardware advances are unlikely to have a significant impact on our ability to solve large, complex optimization problems. These irregular problems require the use of suitable semi-optimal algorithms that may trade some amount of solution quality for substantially reduced execution times. Scalable algorithms are also preferred so we can take full advantage of emerging parallel computer architectures to further reduce execution time and/or solve larger problems. This thesis effort evaluates the performance and scalability of one class of semi-optimal algorithms (GAs) applied to a particular irregular problem (the protein folding problem).

This chapter introduced the general research problem, described the main elements of our approach, and rationalized the need to expend the research effort on genetic algorithms and the protein folding problem. Chapter II expands on the current genetic algorithm and parallel processing literature relevant to this work. Chapter III summarizes the knowledge pertinent to the protein folding problem and analyzes the problem space. Chapter IV discusses the design and implementation of genetic algorithms and a force-field energy model. Chapter V outlines the experimental design used to evaluate the performance of genetic algorithms for the protein folding problem. Chapter VI presents and analyzes the experimental data. Finally, Chapter VII draws conclusions from this investigation and highlights promising areas for future research.

II. Genetic Algorithm (GA) Literature Review

This chapter is an introductory discussion of genetic algorithms (GAs). Section 2.1 provides a historical context for evolutionary algorithms. Sections 2.2 through 2.4 present the theory and mechanics of simple GAs (SGAs), messy GAs (mGAs), and fast messy GAs (fmGAs) respectively. Finally, Section 2.5 describes techniques used to parallelize genetic algorithms.

Genetic algorithms (GAs) are a stochastic search/optimization technique loosely based on natural evolution and the Darwinian concept of "Survival of the Fittest" (34:1)(49). A generalized genetic algorithm consists of a *population* of *encoded* solutions that are manipulated by a set of *operators* and evaluated by some *fitness function* that determines which solutions survive into the next *generation.*

For our purposes, search and optimization techniques fall into two broad categories: deterministic (a combination of calculus-based and enumerative) and stochastic (random) methods (34:2). Greedy algorithms, calculus-based methods, and tree/graph search techniques are all examples of deterministic approaches (2). These methods have been successfully used to solve a wide variety of problems. However, there is an even greater number of problems that are discontinuous, multi-modal, or NP-complete where deterministic methods fail miserably (34:3–6)(30, 26). The main stumbling block for deterministic methods is their requirement for some amount of problem specific knowledge to direct or limit the search. For example:

1. Greedy algorithms assume optimal sub-solutions are *always* part of the optimal solution (2:80)(52)
2. Calculus-based methods require continuity (56:167)
3. Tree/graph search techniques need problem specific heuristics/decision algorithms to limit the search space (30, 73)

A partial list of problem characteristics that can make deterministic search techniques unsuitable for a particular problem includes: multi-modal and/or discontinuous solution spaces, exponential search spaces (NP-complete problems), and limited domain knowledge (no heuristics). Problems that exhibit one or more of these characteristics are called *irregular* (55).

Stochastic search and optimization approaches (simulated annealing, evolutionary strategies, evolutionary programming, genetic algorithms, monte-carlo techniques) have been developed that supplement deterministic techniques (66, 34). The single requirement for stochastic methods is a

function that assigns fitness values to possible solutions. Although these methods cannot guarantee the optimum solution, in general they can provide *good* solutions to a wide range of problems that may be irregular and/or exponentially too large for deterministic methods (34:6–7)(52).

This chapter reviews the current literature on genetic algorithms starting with a short historical perspective of evolutionary algorithms in section 2.1. Sections 2.2, 2.3, and 2.4 discuss the rationale and mechanics of simple GAs, messy GAs, and fast messy GAs respectively. Finally, section 2.5 summarizes the current state of parallel genetic algorithms.

2.1 Brief History of Evolutionary Algorithms

Evolutionary models based on natural selection and genetic theory started appearing during the late 1950s and early 1960s. The first working models were computer simulations of genetic and biological systems. The idea of using algorithms that model natural evolution as search and optimization techniques began in the late 1960s and 1970s (34:89–104). A class of methods called evolutionary algorithms (EAs) is composed of three main categories of investigation based on that early work: Evolutionary Programming (EP), Genetic Algorithms (GAs), and *Evolutionstrategie* (Evolution Strategies, ESs) (34:104–106)(1, 13). However, this taxonomy is not universally accepted. Many authors categorize only EP and ESs as Evolutionary Algorithms, and put GAs in a separate category by themselves (27:24). Even though simulated annealing (SA) is based on thermodynamics, it's often associated with evolutionary algorithms because it uses a mutation operator (81:17).

Fogel, Owens, and Walsh were the first to propose the technique known as evolutionary programming. Evolutionary programming tries to generate artificial intelligences by an evolutionary process that allows the survival of organisms that respond appropriately to a given environment. It has been applied to problems such as sequential symbol prediction and process control. EP usually operates on the components of abstractions such as finite state machines or programming languages (1, 26, 28).

Evolution strategies is an algorithm conceived by Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin while they were searching for optimal airfoil shapes. The original formulation consisted of a one-member population that was operated on by mutation only.

The representation consisted of a pair of real-valued vectors ($V = \langle \mathbf{x}, \sigma \rangle$) where the first vector, $\mathbf{x}$, represents a solution and the second vector, σ, is a vector of standard deviations. The mutation operator creates a new individual using a normal distribution with zero mean as follows:

$$\mathbf{x}^{t+1} = \mathbf{x}^t + N(0, \sigma) \tag{2.1}$$

Various refinements have been made to this original design including population sizes greater than one, recombination operators, and dynamic changes to the vector σ (66:128–132)(1).

John Holland's work on adaptive systems is recognized as the fundamental beginning of genetic algorithms. Previous researchers had used computers to simulate evolutionary systems, and Fraser even tried to optimize a phenotype function, but nobody before Holland recognized the role that nature's evolutionary process could play in search and optimization (34). The embarkation point for all GA research is Holland's *"Adaptation in Natural and Artificial Systems"* (48) which established the mathematical basis for GAs, including the then unnamed *Schema Theorem* or *Fundamental Theorem of Genetic Algorithms*, and generalized schemes for reproduction, crossover, mutation, and inversion (34:89–92).

Three other names have come to be synonymous with GA research: Kenneth A. De Jong, David E. Goldberg, and John J. Grefenstette. De Jong's dissertation (14) put Holland's theory to the test and introduced GA infrastructure that is still in use today (a suite of test functions and performance measures). Although his dissertation focused on function optimization, most of his work to date concerns his broader interest in machine learning (79, 15, 16). Goldberg began by applying genetic algorithms to machine learning and optimization problems. His most recent efforts include work on optimal GA population sizes and alternate GA paradigms (messy GAs and fast messy GAs) to combat deceptive problems (34:387–389)(38, 36). Grefenstette is probably best known for his genetic algorithm implementation, GENESIS, which has been used as a basic GA workbench by many researchers (44). He has also worked on optimal GA parameter sets and machine learning using genetic algorithms (42, 46, 43, 45).

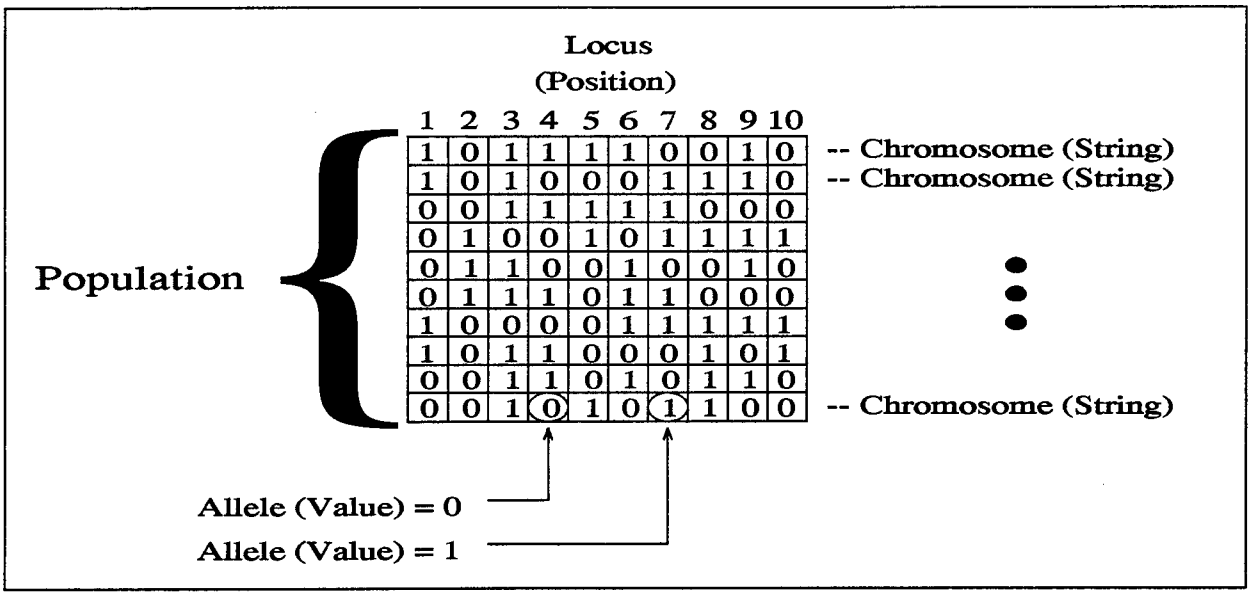


Figure 2.1 Simple Genetic Algorithm Data Structures and Terminology

2.2 Simple Genetic Algorithm (SGA)

Simple genetic algorithms are based on theories of natural genetics and therefore share some of the same terminology. Figure 2.1 illustrates the following terms. A *string* or *chromosome* contains *genes* that encode a solution to a particular problem. Each gene has a *locus* and *allele* associated with it. The locus, or position of a gene generally determines its meaning. A gene expresses only one value or allele at a time from a set of values that are allowed for that gene. A bag of strings is called a *population.* A genetic algorithm *evolves* populations toward better solutions of the encoded problem.

Most SGA implementations use strictly binary encodings of the problem parameters, usually in a form such that x_{min} corresponds to a string of all 0's, x_{max} corresponds to a string of all 1's and there is a linear mapping of all values between x_{min} and x_{max}. Some problems may benefit from the use of gray code parameter encoding (34:101). In gray code, the encoding of successive integers differ by a single bit (54:40). Higher cardinality encodings have also been investigated for certain other problems, most notably combinatoric problems (82, 70). The problem encoding is a very important design decision in the formulation of a genetic algorithm to solve a specific problem.

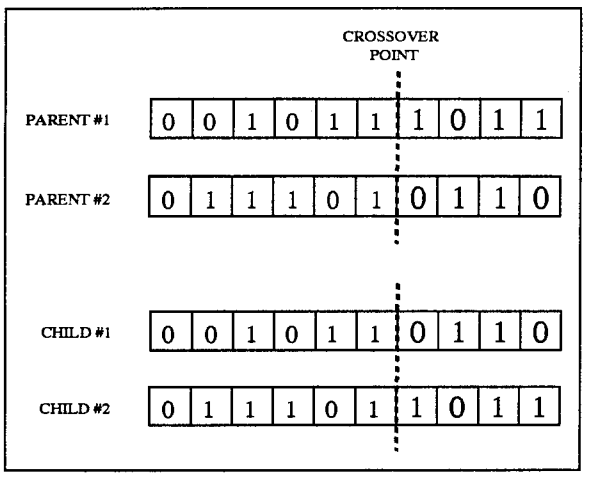


Figure 2.2 Single-Point Crossover

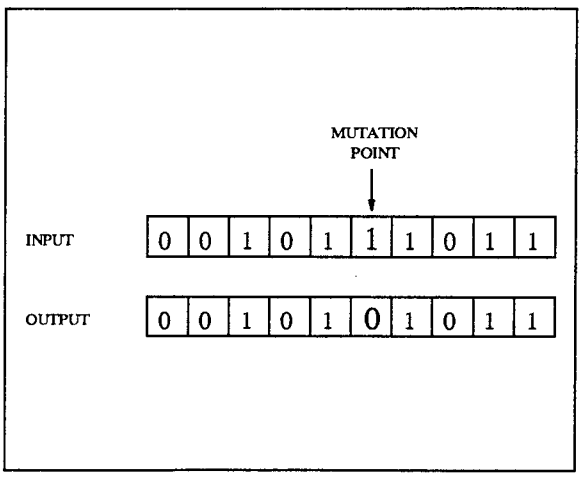


Figure 2.3 Bitwise Mutation

2.2.1 Simple Genetic Algorithm Operators. The three standard operators associated with simple genetic algorithms are *selection*, *crossover*, and *mutation* (66:21,56)(34:10)(81). Above-average individuals of a population are *selected* to become members of the next generation more often than below-average individuals. *Crossover* recombines pieces of solutions to test different combinations of existing solutions. In the absence of other operators, selection and crossover will eventually force a population of solutions to *converge* to a single solution (34:14)(40). *Mutation* is an operator designed to encourage diversity in a population so that convergence occurs more slowly and more of the solution space can be explored.

Figures 2.2 and 2.3 illustrate the function of *single-point crossover* and *bitwise mutation* respectively on binary encoded strings that are ten bits long. Crossover operates on two strings, called the parents, to create two new strings, called the children. After two parents and a crossover point have been arbitrarily chosen, the bits after the crossover point are exchanged to create the children. Mutation is a unary operator that takes one string as input, arbitrarily chooses a bit position within the string, and changes the bit in that position to the opposite value. Many other crossover and mutation operators are possible, and indeed necessary, to exhibit different recombination characteristics and operate on alternate encodings (82, 88).

Figure 2.4 represents the operation of a *proportional* selection operator, called *roulette-wheel* selection, on two different populations of four strings each. Each string in the population is assigned a portion of the wheel proportional to the ratio of its fitness and the population average fitness.

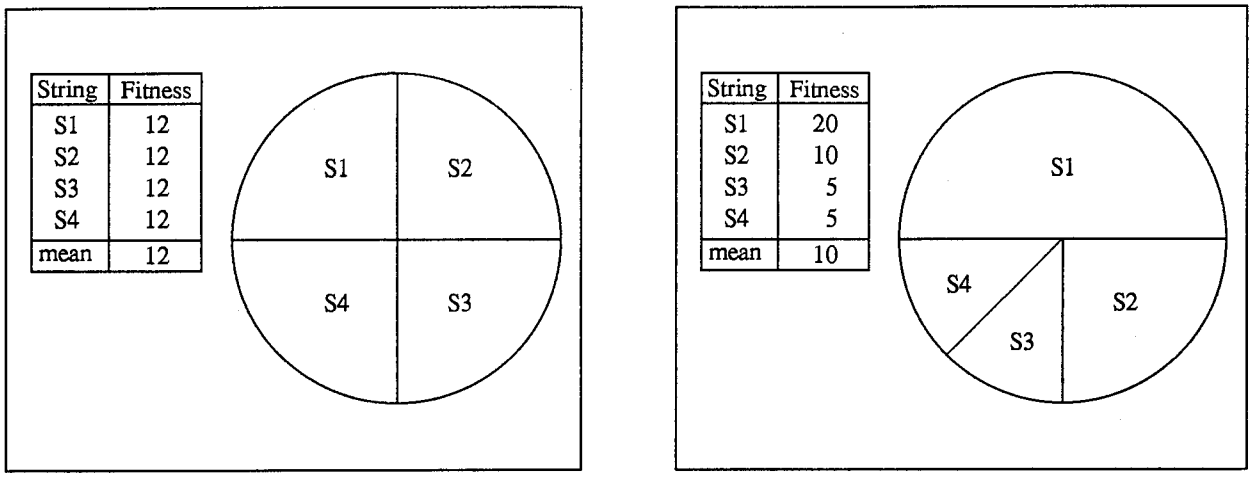


Figure 2.4 Roulette Wheel Selection

1. randomly generate initial population
2. evaluate fitness of all population members
   for i = 1 to the maximum number of generations
3. perform selection
4. perform crossover
5. perform mutation
6. evaluate fitness of all population members
   end loop

Figure 2.5 Psuedo Algorithm for Simple GAs

In the first case where the fitnesses are equal, each string is given an equal share of the wheel (it is equi-likely that any of the four strings will be selected into the next generation). In the second example, $S1$ is twice as likely to be selected into the next generation as $S2$, which is twice as likely to be selected into the next generation as either $S3$ or $S4$. As with crossover and mutation, many other selection operators are possible, each with its own characteristic effect on convergence. *Rank-based* and *tournament* are notable selection operators, and the *elitist* strategy is a modification that can be used with any selection operator (81, 34, 87).

The three operators (crossover, mutation, and selection) and an evaluation function are assembled according to the psuedo algorithm shown in Figure 2.5 to create a simple genetic algorithm.

2.2.2 Simple Genetic Algorithm Parameters. The most difficult part of genetic algorithms is selecting a parameter set that will generate the best performance (efficiency and effectiveness). Efficiency is a measure of the computer resources (cpu time, memory) required to obtain a solution. Effectiveness relates the solution quality of various algorithms. Both measures are relative to the specific problem being tackled and tradeoffs can be made between the two.

Some interrelated SGA parameters include: population size, crossover probability, and mutation probability. Theoretical analysis and empirical studies have been accomplished to formulate estimates of what each of these parameters should be to encourage a robust search that terminates with a near-optimal solution (35, 77). While these two particular studies seem at odds with each other, their conclusions are based on entirely different measurements of GA progress. Schaffer's empirical study is aimed at maximizing *on-line performance* or progress toward optimal solutions (efficiency) without regard for the final solution. Goldberg's theoretical work makes conservative choices to establish confidence levels for the optimality of a final solution (effectiveness) and ignores the astronomical resource costs required to achieve it!

Many relationships have been observed between parameter settings and performance. Increasing the population size generally improves the final solution, however the increase in execution time becomes prohibitive (14, 33). Population size has been shown to exhibit an inverse relationship with mutation rate and, to a lesser extent, crossover rate (77:55). Although there is no evidence so far of any correlation between crossover and mutation probabilities it is generally accepted that using crossover without mutation is insufficient for a robust search (52, 78, 23). However, it has been postulated (especially from the other branches of evolutionary algorithms) that mutation is the only necessary operator (27, 78). Other researchers are examining the effects of changing parameter settings during GA execution, either on some predefined schedule or possibly by monitoring GA metrics during the run (11, 25).

2.2.3 Mathematical Theory of How (Why) Simple GAs Work. *Schemata* are templates that define sets of strings with the same values at certain string positions and are represented using an additional *don't care* symbol (*) (34:19,29). For example, the schema *101 represents the set of strings {0101, 1101} and the schema 1*0*01 defines the set {100001, 100101, 110001, 110101}. The *defining length* ($\delta(H)$) and *order* ($o(H)$) are two values associated with a particular schema H. The

defining length of a schema is a measure of the distance between the first and last fixed positions. The order of a schema is the number of positions with fixed values. Using the sample schemata from above $\delta(*101) = 4 - 2 = 2$, $o(*101) = 3$, $\delta(1*0*01) = 6 - 1 = 5$, and $o(1*0*01) = 4$.

The Schema Theorem, represented by

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m\right], \tag{2.2}$$

establishes a lower bound on the number of representatives schema H will have in the next generation ($m(H, t+1)$) based on the:

1. number of representatives in the current generation ($m(H, t)$),
2. fitness of schema H vs the population average fitness ($\frac{f(H)}{\bar{f}}$),
3. string length, defining length, and probability that schema H will be destroyed by crossover ($p_c \frac{\delta(H)}{l-1}$), and
4. order and probability that schema H will be destroyed by mutation ($o(H)p_m$).

Although it would appear that genetic algorithms operate only on the specific strings in a population, it has been shown that many of the 2^l schemata in each string are processed simultaneously (*implicit parallelism* (48:71–72)(34:40)). The Fundamental Theorem of Genetic Algorithms (Schema Theorem) states that all schemata will receive representation in the next generation proportional to the ratio of their fitness to the average fitness of the population. This representation is reduced by the amount of disruption that crossover and mutation can cause to a schema. More succinctly,

> short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations (34:33).

2.2.3.1 Complexity Analysis. Using the standard SGA operators, the time complexity of selection, crossover, and mutation are generally $\mathcal{O}(nl)$ where n is the population size and l is the string length. Given a fixed number of generations, SGA execution time is $\mathcal{O}(nl)$ (18). However, the time complexity of the fitness function for real-world problems (expressed as some function of the string length and problem space parameters) usually dominates the time to execute the genetic algorithm control sequence (67), therefore great care should be taken in its analysis and design.

It is easy to see that the space complexity of a simple genetic algorithm is also $\mathcal{O}(nl)$ because the population of solutions has to be stored. However, the space complexity can also be affected by the data structure requirements of specific problems. For example, a function that relies on a lookup table to evaluate solutions will require more space. If that space is related to the problem size in a way that make the lookup table grow faster than the population size and string length, then the problem space requirements dominate the space complexity of the entire algorithm.

2.3 Messy Genetic Algorithm (mGA)

Genetic algorithms are designed to take advantage of the *building block* theory (48, 34, 29). The main idea is that small pieces of a solution which exhibit above average performance are combined to create larger pieces of above average quality, which are themselves recombined into larger pieces, and so forth.

Simple genetic algorithms suffer from the fact that the "pieces" that form the building blocks must be put next to each other explicitly in the fixed encoding or else they are more likely to be disrupted by crossover. This problem is magnified when competing schemata (schemata with different values at similar defining positions) define locally optimal solutions. *Deception* occurs when a locally optimal building blocks are selected instead of globally optimal ones. Messy genetic algorithms (mGAs) were designed to deal with these problems by encoding the string position (locus) along with its value (allele). This gives a messy genetic algorithm the ability to search for the "true" building blocks of the problem and create tighter *linkage* for those genes than a fixed position encoding would allow (39). The mGA encoding scheme also allows *under-specified* and *over-specified* strings to exist in the population. Under-specified strings don't have an allele defined for every locus and are evaluated with the aid of a locally optimal *competitive template* that supplies values for the unspecified genes. Over-specified strings contain multiple alleles specified at the same locus and are processed in a left-to-right fashion which sets the gene to the value encountered first. The desire to create and manipulate superior building blocks is the motivation behind messy genetic algorithms (39, 37, 38).

2.3.1 Messy Genetic Algorithm Operators. Messy GAs use variations of the same genetic operators used by simple GAs. In the few implementations of mGAs that exist (38, 39, 37, 62),

tournament selection has been used instead of proportional or rank-based selection because of its desirable performance characteristics (38:50)(35, 24). The tournament selection operator also has a *thresholding* mechanism added to it which ensures that strings have a number of positions in common before competition is allowed (37:424–427). Crossover is replaced by a combined *cut-and-splice* operator that works on variable length strings. As the names suggest, *cut* divides a string into two smaller pieces and *splice* concatenates two strings to form a single, longer string. A mutation operator that can change a gene's value or its position has been described but unused in any mGA implementations (39:504).

Messy GAs employ a different initialization strategy compared to SGAs. The main processing loop of an mGA is composed of *primordial* and *juxtapositional* phases. During *partially enumerative initialization (PEI)*, exactly one copy of each possible building block of the specified size (k) is generated. Thus, the initial population size for a messy GA is generally quite large ($2^k \binom{l}{k}$) (39:420). The primordial phase serves two basic purposes: enrich the population with above average building blocks and reduce the population to a size that can be efficiently and effectively processed by the juxtapositional phase. Tournament selection, the only active operator during the primordial phase, fills the population with above average building blocks, then periodically the population size is halved. No additional fitness evaluations are required during the primordial phase. The juxtapositional phase is most similar to the main processing loop of a simple GA (39:506). Cut-and-splice and any other genetic operators are applied, fitness evaluations are performed on the newly created strings, and tournament selection bolsters the next generation with highly fit solutions. A psuedo algorithm for messy GAs is shown in Figure 2.6.

2.3.2 Messy Genetic Algorithm Parameters. The major parameter settings associated with messy GAs are population size, cut-and-splice probabilities, and a schedule for reducing the population size. Initial population size can be calculated once the string length and block size have been determined. String length is simply a function of the encoding used, but block size is a problem dependent quantity that may be difficult to estimate. The final population size at the end of the primordial phase is even less quantifiable! The splice probability is consistently set to 1.0 with the following rationale: the primordial phase ends with a population of optimal building blocks which should only require assembly to form a complete string that is a near-optimal solution

```
1.    perform partially enumerative initialization
      evaluate fitness of all population members
2.    for i = 1 to the maximum number of primordial generations
          perform tournament selection
          if (a suitable number of generations have transpired) then
              reduce the population size
          end if
      end loop
3.    for i = 1 to the maximum number of juxtapositional generations
          perform cut-and-splice
          perform other operators (currently not used)
          evaluate fitness of all population members
          perform tournament selection
      end loop
```

Figure 2.6 Psuedo Algorithm for Messy GAs

(38:25). The chosen cut probability is scaled by the current length of a string so that longer strings are more likely to be cut than shorter strings. The schedule for reducing population size during the primordial phase typically allows for two or three generations of enrichment followed by cutting the population in half (39:505). No theoretical or empirical work has been accomplished to provide any guidance for final primordial population size, cut probability, or population reduction schedules for messy GAs.

2.3.3 Mathematical Theory of How (Why) Messy GAs Work. The Schema Theorem (Equation 2.2) is directly applicable to messy genetic algorithms. The rationale for messy genetic algorithms follows from the theorem's interpretation: "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations." If the building blocks of a problem aren't encoded as short, low-order schemata then crossover and mutation will disrupt the formation of those building blocks.

For problems using a fixed encoding, where the identification of building blocks is prohibitive or impossible, Goldberg has calculated the *normalized expected defining length* ($\frac{\langle\delta\rangle}{l+1}$) for k-sized building blocks (Equation 2.3). The normalized expected defining length is a measure of the mean length of the schemata that make up the building blocks of a randomly encoded problem. The

interpretation is that an arbitrary encoding is highly unlikely to establish tight linkage for the building blocks of a problem (39:498–499).

$$\frac{\langle \delta \rangle}{l+1} = \frac{k-1}{k+1} \tag{2.3}$$

Messy genetic algorithms take advantage of the Schema Theorem by searching for both the defining positions and gene values of the building blocks using PEI and the primordial phase. Then the juxtapositional phase of the messy GA starts with "short, low-order, above-average" schemata that are also "short, low-order, above-average" building blocks!

2.3.3.1 Complexity Analysis. Because of the partially enumerative initialization (PEI), the time complexity of messy GAs is $\mathcal{O}(l^k)$. This compares unfavorably with the rest of the algorithm which is only $\mathcal{O}(l \log l)$ (37:420–422). Space complexity remains unchanged from simple genetic algorithms. However, the constant term is generally larger and the population size (n) is *much* larger! As is the case with simple genetic algorithms, the time complexity of the evaluation function usually dominates that of the control sequence.

2.4 Fast Messy Genetic Algorithm (fmGA)

The advantage messy GAs have over simple GAs is the ability to create tightly linked building blocks for the optimization of deceptive problems. The disadvantage associated with this better processing is the time complexity of the initialization phase which dominates the mGA algorithm (37:422). Fast messy GAs are a messy GA variant designed to reduce the complexity of the initialization phase, and thus the overall algorithm time and space complexity (36:59).

2.4.1 Fast Messy Genetic Algorithm Operators. PEI and the selection-only primordial phase of mGAs are replaced by *probabilisitically complete initialization* (PCI) and a primordial phase consisting of selection and building block filtering (BBF) in fmGAs. PCI and BBF are an alternate means of providing the juxtapositional phase with highly fit building blocks (36:59–61).

PCI is used to create an initial population whose size is equivalent to the population size at the end of the primordial phase of mGAs. The length of these strings is typically set to $l - k$. The

```
1.    perform probabilistically complete initialization
      evaluate fitness of all population members
2.    for i = 1 to the maximum number of primordial generations
          perform tournament selection
          if (a building block filtering event is scheduled) then
              perform building block filtering
              evaluate fitness of all population members
          end if
      end loop
3.    for i = 1 to the maximum number of juxtapositional generations
          perform cut-and-splice
          perform other operators (currently not used)
          evaluate fitness of all population members
          perform tournament selection
      end loop
```

Figure 2.7 Psuedo Algorithm for Fast Messy GAs

primordial phase then alternately performs several tournament selection generations to build up copies of highly fit strings followed by BBF to reduce the string length toward the building block size (k). Building block filtering is a simple process that randomly deletes several genes from a string. The juxtapositional phase is the same as in mGAs. A psuedo algorithm for fast messy GAs is shown in Figure 2.7.

2.4.2 Fast Messy Genetic Algorithm Parameters. Fast messy GAs need a building block filtering and thresholding schedule instead of the population size reduction schedule required by mGAs. Goldberg provides formulas for deriving schedules (36:60–61), but the formulas contain additional parameters and no guidance is given for choosing their values. The remainder of mGA parameters are used by fmGAs as well.

2.4.3 Mathematical Theory of How (Why) Fast Messy GAs Work. Fast messy GAs are governed by the Schema Theorem (Equation 2.2) just like mGAs. The difference relates to how the population of "good" building blocks is created for processing by the juxtapositional phase. Goldberg performs a detailed analysis to show that a much smaller initial population of long strings

(PCI) can be manipulated (through BBF) to create a population of "good" building blocks just as effectively as PEI and the primordial phase of mGAs (36:60–61).

2.4.3.1 Complexity Analysis. Reducing the overall time complexity of the algorithm is the main reason for switching from mGAs to fmGAs. PCI and BBF result in a time complexity of $\mathcal{O}(l \log l)$ for initialization and the primordial phase combined (36:61). Thus, the design goal has been met—fmGAs exhibit better efficiency than mGAs ($\mathcal{O}(l \log l)$ vs $\mathcal{O}(l^k)$) and preserve their effectiveness. Space complexity for fmGAs remains unchanged from SGAs and mGAs ($\mathcal{O}(nl)$) and populations can be sized much smaller than mGAs. Again, the time and space complexities of the evaluation function usually dominate those of the control sequence.

2.5 Parallel Genetic Algorithms

There are two major concerns when parallelizing any algorithm: is the parallel algorithm correct and is it faster than the serial version? Correctness is an issue because we have even greater difficulty verifying parallel algorithms than we have for sequential programs (59). Given that the parallel algorithm is correct, speedup is the primary goal of parallelization (7). A tradeoff analysis is generally required to determine if the estimated benefits warrant the expenditure of resources to parallelize an algorithm. There is evidence to suggest that parallelizing genetic algorithms *is* worthwhile and should be examined further (19, 84, 75, 80, 41).

2.5.1 Decomposition Techniques. Data and control decomposition are alternate means of dividing a problem into portions that can be worked on simultaneously. In general, data decomposed algorithms perform the same operations of subsets of the input (*data parallelism*) and control decomposed algorithms perform different operations on the total input (60). In either case the results are recombined in some fashion to obtain the final result(s). Genetic algorithms are easily parallelized because they are highly data decomposable (although control decomposition is not impossible, especially as the complexity of operators and evaluation functions grow). Parallelizing GAs using data decomposition can be as simple as running multiple copies of the same program on different processors, each starting with a different random number seed, and then choosing the best result from all runs. Data parallelization techniques are also amenable to static load balancing

because their computation and communication patterns are regular (22, 60). This does not imply that all processors are searching in equally promising portions of the search space. Thus, some efficiency may be lost to subpopulations that are searching similar solution neighborhoods or stuck in local optima. Two models, which lie at opposite ends of a granularity spectrum, have been proposed for parallel genetic algorithms—the *island model* (course-grained) and the *neighborhood model* (fine-grained) (18, 41). These models are designed to improve the simplistic parallel approach by sharing near-optimal results with some portion of the global population.

2.5.2 Island and Neighborhood Model. The island model is an extension of the simplistic approach where the total population is divided into subpopulations which are distributed among the processors. The subpopulations evolve in parallel, however at certain time intervals a *migration* occurs where solutions are communicated between processors (18:10). Migration rates, migration selection strategies, and migration patterns are additional parameters with associated design decisions that must be defined for the parallel genetic algorithm. Near-linear speedup is expected and has been observed for island model parallel genetic algorithms (3:60)(84, 5). The time complexity of island model GAs is $\mathcal{O}(\frac{nl}{p})$, where p is the number of processors, n is the population size, and $p \ll n$ (18). As $p \rightarrow n$, the resulting small subpopulation size increases the ratio of commmunication time to compute time and the speedup becomes much less than linear. Island model GAs are typically used on course-grained or multiple-instruction-multiple-data (MIMD) architectures (52).

The neighborhood model splits the population up spatially in a two- or three-dimensional grid. This grid and the definition of a neighborhood limits the interaction of individuals in the population. Typically, a single string is assigned to each processor, therefore crossover and selection must be modified because their operation is distributed across more than one processor. Although their convergence characteristics have been observed to be better than the Island Model (18:24), neighborhood model parallel genetic algorithms don't exhibit speedup because they assume $n = p$, therefore no speedup can be obtained because more/fewer processors are not allowed. The neighborhood model is most often compared to the simple GA for time complexity analysis ($\mathcal{O}(s + l)$ vs $\mathcal{O}(nl)$ where s is the neighborhood size) (18:24). Neighborhood model GAs are generally implemented on fine-grained or single-instruction-multiple-data (SIMD) architectures (18).

2.6 Summary

Genetic algorithms are semi-optimal search/optimization techniques capable of finding "good" solutions to problems that are intractable for deterministic methods. Many theories and conjectures have been proposed based on both mathematical analysis and toy problem experimentation with genetic algoirthms, their control structure, and different genetic operators. Genetic algorithms have been applied to some relatively small real-world problems with good results. The question is, can the theory and empirical evidence stand up to a large-scale, real-world application? The next chapter provides the background for a specific application of genetic algorithms to a difficult problem in biochemistry.

III. The Protein Folding Problem Literature Review

This chapter provides the background required to understand the mechanics and ramifications of the protein folding problem. Section 3.1 defines terminology in the biochemistry domain. Section 3.2 describes the expensive experimental techniques used to determine the structure of proteins. Finally, Section 3.3 examines various models used to predict the structure of polypeptides and proteins.

The protein folding problem (PFP) has been recognized as a National Grand Challenge problem in biochemistry and high-performance computing (10). The challenge is to find a method to predict the three-dimensional topology of a protein based on the sequence of its components. A solution, which would provide knowledge about the function(s) of individual proteins, is also the first step toward solving the *inverse folding problem* (IPFP) (6, 58). The inverse folding problem is to determine a sequence (possibly more than one) that will fold to a specified three-dimensional structure.

The difference between the two problems is best characterized by the ability a solution to either would provide: a PFP solution would enable the *evaluation* of many proteins in a search for one with a specific property or function; an IPFP solution would provide a mechanism to *design* a protein with specified characteristics (6:25–26). Possible applications include: pharmaceuticals with few or no side effects; energy conversion and storage capabilities (similar to photosynthesis); biological and chemical catalysts and regulators; angstrom scale information storage; and possible optical/chemical shielding from harmful radiation sources (6:25)(58:5)(71).

3.1 Introduction to Proteins and Associated Terminology

Proteins (polypeptides) are linear sequences of the 20 naturally occurring amino acids. Each amino acid consists primarily of three common *backbone* atoms (a nitrogen and two carbons [$N-C_\alpha-C_\gamma$]) and a distinct combination of atoms and covalent bonds, called the *side-chain* (S_i), connected to the C_α carbon atom. A particular protein is defined by a unique amino acid sequence known as the *primary structure* of the protein (6:24)(58:2)(57:49). Figure 3.1 depicts a generic protein composed of three amino acids. The primary structures of approximately 50,000 proteins

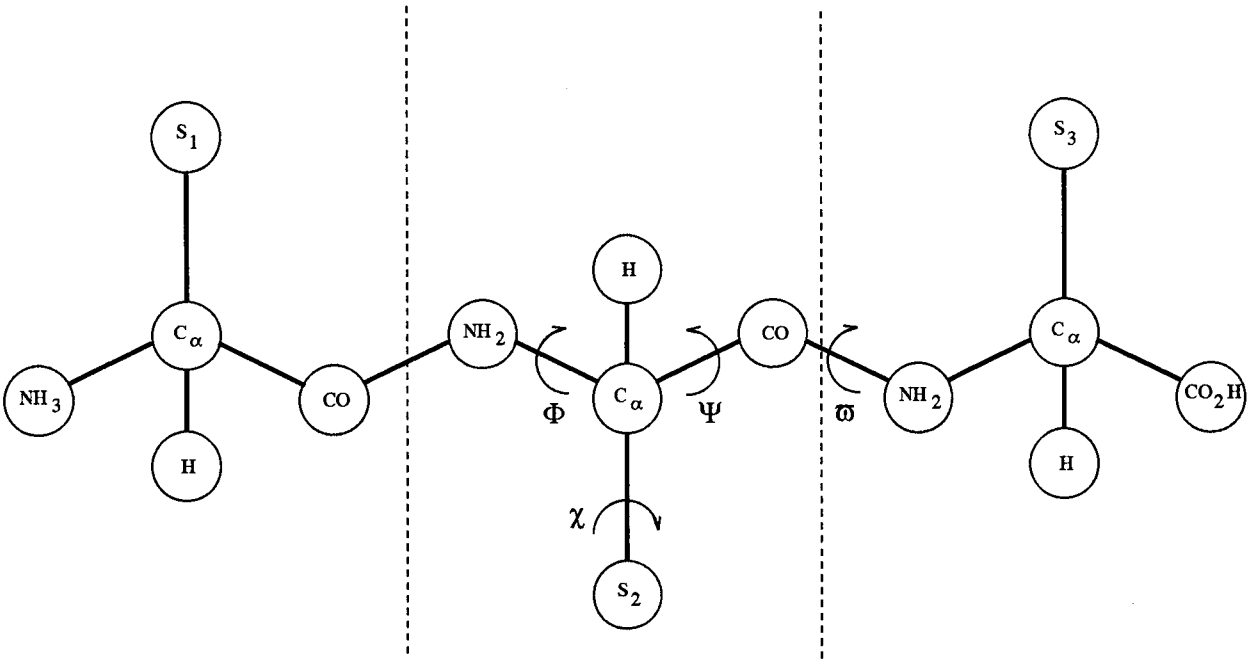


Figure 3.1 A Three Amino Acid Protein

are currently known and this number is expected to double every year, due largely to the Human Genome Project and the ease with which sequences are experimentally determined (58:5)(69).

There are a number of features that subsequences of proteins exhibit regularly. *α-helices* and *β-sheets* are two common examples of local features that are called the *secondary structure* of a protein (6:24). Secondary structures are used primarily as a means of classifying the local geometric arrangement of proteins and some researchers are investigating the utility of predicting secondary structure as the first step of tertiary structure prediction (57:50).

The three-dimensional structure of a protein is the major determinant of its function. This three-dimensional shape is called the *tertiary structure* or *conformation* of the protein. Proteins assume their *native* conformation, which is unique and compact, in their natural biological environment (typically in aqueous solution, at neutral pH and 20–40° C)(6, 58). A protein in its native conformation is only slightly more stable than the various conformations with marginally higher energies. This single fact is responsible for the major difficulty of the protein folding problem (6:24–25)(58:2–4)(57:50).

Table 3.1 Enumeration Time of a 1.3×10^{30} Search Space at One Solution per Clock Cycle

Computer Speed	Execution Time (years)
1 GigaFLOP	$\approx$ 41 trillion
1 TeraFLOP	$\approx$ 41 billion
1 PetaFLOP	$\approx$ 41 million

The position of all atoms in a protein can be determined given the position of one atom, the *bond length* of each covalently bonded pair of atoms, the *bond angle* formed by each triplet of bonded atoms, and the *dihedral angle* formed by each bonded group of four atoms (see Figures 3.2 - 3.4). Given this set of parameters, every protein has $3n - 6$ degrees of freedom where n is the number of atoms. However, the bonds and bond angles are relatively rigid, therefore the independent dihedral angles are left as the only dominant factor to determine the tertiary structure of a protein and the degrees of freedom are reduced by a factor of approximately 2/3 (6:26)(57:50). Each amino acid contains a ϕ, ψ, and ω dihedral angle and zero or more χ_i dihedral angles as shown in Figure 3.1.

If we discretize the domain of the dihedral angles so that there are d possible values, then the size of the search space is given by d^N where N is the number of independently variable dihedral angles. Given a very coarse 20° discretization of the $0 - 360°$ dihedral angle domain and a small protein with 24 independently variable dihedral angles, the search space contains $18^{24} \approx 1.3 \times 10^{30}$ conformations. Table 3.1 shows the time required to enumerate the search space on current and envisioned high performance computers (under the optimistic assumption of one evaluation per clock cycle)(83:7)! (Giga-, Tera-, and Peta-FLOP computers can perform $10^9, 10^{12}$, and 10^{15} floating point operations per second respectively.) Therefore, if we hope to find the single native conformation of a protein, we must have access to efficient search algorithms that severely prune the search space.

3.2 Experimental Tertiary Structure Determination

In comparison with the number of known protein sequences, the number of known native conformations is extremely small (approximately 400). The tertiary structures of these proteins have been determined experimentally using either X-ray crystallography or nuclear-magnetic-resonance (NMR) spectroscopy. These techniques are inadequate for the task because they take one to two years to obtain results for a single protein (6:25)(58:5).

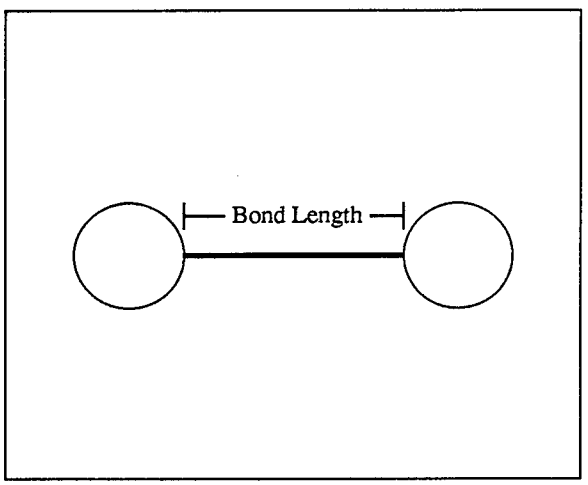


Figure 3.2 Protein Bond Length

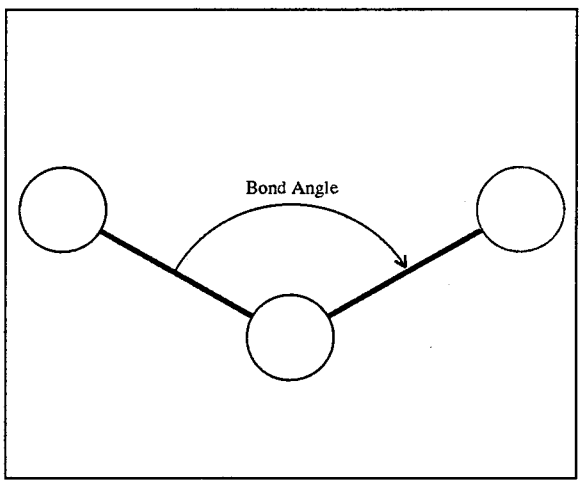


Figure 3.3 Protein Bond Angle

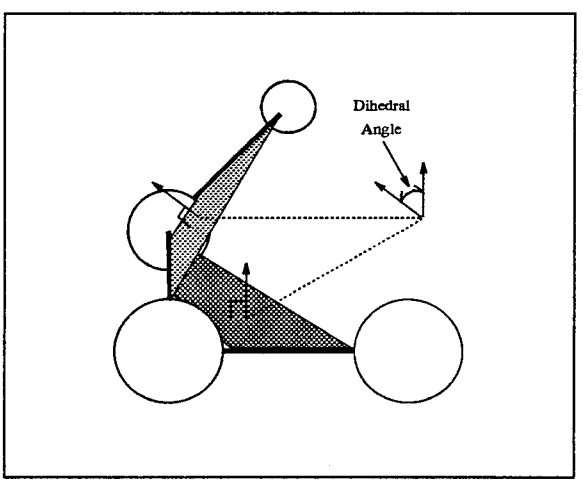


Figure 3.4 Protein Dihedral Angle

3.3 Tertiary Structure Prediction (PFP)

To reduce the gap between the number of known protein sequences and native conformations, we need to be able to reliably predict the tertiary structure of proteins in a reasonable amount of time. *Exact* versions of the classical methods discussed next are theoretically capable of finding the native tertiary structure of any protein. In practice, the computational cost of the calculations prohibits the use of these exact methods. The classical methods that are computationally viable are typically relaxed formulations that ignore the high-order interaction terms. The applicability of the other prediction methods discussed below is severely limited.

3.3.1 Classical Prediction Methods. *Molecular dynamics* is a technique that attempts to simulate the protein folding process. The protein is treated as an N-body simulation and Newton's motion equations are solved to determine the location of all the atoms at discrete points in time. Molecular dynamics faces two major difficulties in its attempt to fold proteins. First, the number of atoms that must be simulated is very large:

1. Small proteins contain hundreds of atoms.
2. Larger proteins can be composed of several ten-thousands of atoms.
3. Thousands of atoms must be added to simulate the surrounding solution.

Second, the thermal oscillations of bonded atoms have a period between $10^{-14} - 10^{-13}$ seconds. Simulation time steps in the femtosecond (10^{-15} sec) range are required to accurately account for these harmonics. These two factors have limited molecular dynamics simulations to less than a few nanoseconds (10^{-9} sec), even on today's fastest supercomputers. That time-frame is ten orders of magnitude too short to simulate the folding process of most proteins (6:27)(58:6–7). Using an *extended-atom representation* is one method that can greatly reduce the impact of these two problems. The extended-atom representation combines hydrogen atoms with the heavier atoms they are bonded to. This representation generally halves the number of "atoms" in the problem and allows the size of the simulation time steps to be increased (4:189).

The *energy minimization* approach assumes that proteins, like other physical systems, assume that state which minimizes total energy in the system (however, this assumption is not universally accepted (51)). There are three types of energy minimization methods that differ by their time complexity and the accuracy of their calculations. *Ab initio* methods calculate the energy exactly.

Table 3.2 Time Complexity of Energy Minimization Methods

Energy Calculation Method	Time Complexity	Time Estimate for $n = 1000$
ab initio	$\mathcal{O}(n^5)$	11.5 days
semi-empirical	$\mathcal{O}(n^4) - \mathcal{O}(n^3)$	17 min - 1 sec
force-field	$\mathcal{O}(n^2)$	1 msec

Semi-empirical methods eliminate the non-dominating interaction integrals from the calculation. *Force-field* methods simply account for the pairwise interactions between atoms with an appropriate parameterization that implicitly accounts for multi-particle interactions (58:6). Table 3.2 compares the time complexity of these three energy minimization models and gives example execution times for a moderately sized protein ($n = 1000$) assuming the individual component calculations take one nanosecond (10^{-9} sec).

CHARMM, AMBER, and ECEPP are three example of force field energy models that are based on Equation 3.1 (57). $K_{r_{ij}}$, $K_{\Theta_{ijk}}$, $K_{\Phi_{ijkl}}$, r_{eq}, Θ_{eq}, n_{ijkl}, γ_{ijkl}, A_{ij}, and B_{ij} are empirical constants supplied as input. $\mathcal{B}$, $\mathcal{A}$, $\mathcal{D}$, and $\mathcal{N}$ represent respectively the sets of: bonded atoms, atoms forming bond angles, atoms forming dihedral angles, and non-bonded atoms. All atoms with more than three bonds separating them are considered non-bonded. All three models use the non-bonded term explicitly. ECEPP only calculates bonded and bond angle terms around disulfide bridges and dihedral terms for the independently variable dihedral angles. CHARMM implements Equation 3.1 "almost" verbatim with additional constraints on the range over which terms are computed. AMBER adds a partial non-bonded calculation to the dihedral term plus an extra term to handle polar hydrogen non-bonded interactions with nitrogen and oxygen (Equation 3.2) (57:50–53)(4:190–193).

$$
\begin{aligned}
E \;=\; & \sum_{(i,j)\in\mathcal{B}} K_{r_{ij}}(r_{ij} - r_{eq})^2 \\
& + \sum_{(i,j,k)\in\mathcal{A}} K_{\Theta_{ijk}}(\Theta_{ijk} - \Theta_{eq})^2 \\
& + \sum_{(i,j,k,l)\in\mathcal{D}} K_{\Phi_{ijkl}}[1 + \cos(n_{ijkl}\Phi_{ijkl} - \gamma_{ijkl})] \\
& + \sum_{(i,j)\in\mathcal{N}} \left[\left(\frac{A_{ij}}{r_{ij}}\right)^{12} - \left(\frac{B_{ij}}{r_{ij}}\right)^{6} + \frac{q_i q_j}{4\pi\varepsilon r_{ij}} \right]
\end{aligned}
\tag{3.1}
$$

3.3.2 Other Prediction Methods. Structure prediction by *homology* attempts to align the sequence of a protein with an unknown tertiary structure with one whose native conformation is known (58). It has been observed that if the sequences are similar, then the conformations are nearly identical. An extension of homology, called *sequence-structure alignment*, builds a partial monotonic mapping directly from the sequence of the unknown protein to the known tertiary structure of the similar protein. The differences between the two structures are usually surface characteristics built upon the same core structure. Both of these methods are severely limited by our tiny database of currently known protein structures. They are also incapable of predicting the native conformation of proteins with novel structures (58:7–9).

$$
\begin{aligned}
E \;=\; & \sum_{(i,j)\in\mathcal{B}} K_{r_{ij}}(r_{ij}-r_{eq})^2 + \sum_{(i,j,k)\in\mathcal{A}} K_{\Theta_{ijk}}(\Theta_{ijk}-\Theta_{eq})^2 \\
& + \sum_{(i,j,k,l)\in\mathcal{D}} \left[K_{\Phi_{ijkl}}[1+\cos(n_{ijkl}\Phi_{ijkl}-\gamma_{ijkl})] \right. \\
& \left. + \left(\frac{A_{il}}{r_{il}}\right)^{12} - \left(\frac{B_{il}}{r_{il}}\right)^{6} + \frac{q_i q_l}{4\pi\epsilon r_{il}} \right] \\
& + \sum_{(i,j)\in\mathcal{N}} \left[\left(\frac{A_{ij}}{r_{ij}}\right)^{12} - \left(\frac{B_{ij}}{r_{ij}}\right)^{6} + \frac{q_i q_j}{4\pi\epsilon r_{ij}} \right] \\
& + \sum_{(i,j)\in\mathcal{H}} \left[\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right]
\end{aligned}
\tag{3.2}
$$

Simplification techniques are used as methods to reduce the conformational search space to a size that will yield to today's algorithmic search strategies (6). *Lattice* models reduce three-dimensional space to a structured grid, where atoms can only be placed on the grid points. The grid is designed to accommodate the typical connections observed in real proteins. Further simplifications have been obtained by reducing or eliminating the explicit representation of side-chains (58:9–10).

3.4 Summary

The determination of the tertiary structure of proteins is a major challenge in biochemistry. Experimental techniques are considered accurate but time consuming, and are incapable of keeping

pace with the number of protein sequences being discovered. Prediction techniques are hampered by the size of the conformational search space and the time complexity of calculating energy or solving motion equations. However, classical prediction methods, combined with novel search and optimization algorithms, show great potential for both a solution to the protein folding problem and a better understanding of the underlying behavior and operation of biological systems. This thesis effort considers the application of genetic algorithms using energy minimization as one such combination for solving the protein folding problem.

IV. Genetic Algorithm (GA) Design and Implementation

Many genetic algorithm designs and implementations exist. Holland established mathematical specifications for families of GAs based on reproductive plans and three genetic operators (crossover, mutation, and inversion) in his original work (48). De Jong implemented a subset of one of those families based on classes of reproductive plans (14). Typically, designs are functionally decomposed to facilitate the construction of GA workbenches. These workbenches are used to study the behavior of many GA operators. Object-oriented designs also exist, and lend themselves to the examination of representation issues and production systems. Table 4.1 lists a few of the implementations that are widely used along with some of their major characteristics.

All of our GA work at AFIT is either directly or indirectly based on the Genesis software package. What follows is a list of the major factors influencing the original and continuing decision to use Genesis as the general GA software platform at AFIT:

1. GA code must be easily portable to multiple serial and parallel hardware platforms.
2. The GA must contain an I/O interface that provides simple parameter input and meaningful progress and convergence output.
3. Our requirements are for a GA workbench to examine the effects of operators and control structures on function optimization and combinatoric search.
4. Our access to Genesis predates access to all other software packages.

Item 1 is the most restrictive, limiting our choices to C and Fortran implementations. Although the original analyses and selection motives have been lost in revisions of the Compendium of Parallel Programs for the Intel iPSC Computers (55), we can hypothesize that Item 4 was a major determinant. Table 4.2 summarizes the software implementations, hardware platforms, and

Table 4.1 Available Genetic Algorithm Implementations

Implementation	Characteristics
Genesis (44)	binary alphabet, functionally decomposed good modularity, written in C command line interface, functional design
OOGA (12)	real-valued and combinatoric representations and operators written in Lisp, object-oriented design
Splicer (NASA)	binary and combinatoric representations and operators written in C, X-Windows interface, functional design

Table 4.2 AFIT Applications and Associated Hardware/Software Platforms (20, 62, 3, 72, 61)

	Sparc Workstation	Hypercube (iPSC/2 or iPSC/860)	Paragon
Genesis (SGA)	Premature convergence[1] Mission routing[2] Protein folding[2]	Premature convergence[1] Communication strategies[1] Mission routing[2] Protein folding[2]	
messy GA	Premature convergence[1]	Premature convergence[1] Population distribution[1]	
fast mGA	Protein folding[2]		Protein folding[2]

associated examinations/applications of genetic algorithms at AFIT. (Theoretical investigations are labeled 1 and applications are labeled 2.)

4.1 GA High-Level Design

The objective of a good preliminary design is to capture the essence of what needs to be accomplished without regard for specific data structures, control flow, or computer architecture. Several paradigms have been proposed specifically for the specification and design of parallel algorithms, including CSP (47), petri-nets (74), and UNITY (7). While these methods can be shown to be equivalent, their expressive powers lie in different dimensions. CSP and petri-nets lend themselves more to the specification of control flow interactions. We choose to use UNITY as our design language because it's better suited to specifying data parallelism at high abstraction levels.

UNITY (Unbounded Nondeterministic Iterative Transformations) is a parallel program specification and design language that separates the description of *what* should be done from *when*, *where*, and *how*. A UNITY program (design) describes *what* assignments need to take place, but a mapping to a specific architecture answers the other three questions (7:8-11). A UNITY program attempts to express the maximum parallelism possible using only assignment statements separated by either parallel bars (||) or a box ([]). (The parallel bars connect assignments that must take place simultaneously. A box separates assignment statements that must be executed at different times.) A time complexity analysis of this maximally parallel design provides a benchmark to evaluate the time complexity realized by various implementations.

Assertions of the form $\{p\}\ s\ \{q\}$ are used in UNITY to indicate that the execution of statement s from a state where predicate p is true results in a transition to a state where predicate q is true (7:40). With the addition of universal and existential quantifiers, and because of the lack of control flow within UNITY programs, this first order predicate logic is sufficient to prove UNITY programs correct. Then the assertions are *properties* associated with the entire program rather than predicates attached to individual pieces of an algorithm. These properties are classified as either *safety* or *progress* properties that define respectively the legal states and the advancement mechanism(s) of an algorithm.

Program development through UNITY proceeds by translating a specification into a high-level design which is then repeatedly refined until sufficient detail is available for implementation. The proof of correctness for an entire program is built as each refinement is proven correct with respect to its parent design.

We have access to specifications (48, 39, 36) and implementations (44, 62, 65) of simple, messy, and fast messy GAs, however the respective designs are either unavailable or inconsistent with the available software products. This section elaborates high-level UNITY designs for the three GAs, discusses mappings to serial and parallel architectures, and establishes their design time complexities.

With a few minor exceptions, the following notational conventions are observed in these genetic algorithm UNITY designs. Capitalized tokens are either boolean flags or input parameters and can be identified by their context—flags are on the left side of assignment statements and parameters are on the right. Thus, the two can be distinguished from their context. Variables are in *italics* and programming constructs are in regular text.

4.1.1 Simple Genetic Algorithm. Figure 4.1 reflects the top-level design of what a simple genetic algorithm should accomplish with the minimum set of constraints on when, where, and how it should be done. Detailed descriptions of what the individual components do are shown in Figures 4.2 through 4.5 for initialization, selection, crossover, and mutation. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. Noting that all the GA operators are specified using the parallel bars (||)

exclusively with no synchronization variables, we can conclude that each operator is also $\mathcal{O}(1)$ given enough processors.

The following properties are evident from the SGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the third assignment statement. The fixed point states that MAX_GENS generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

invariant
$0 \leq gen \leq MAX\_GENS + 1$
FP $\equiv$
$\langle gen = MAX\_GENS + 1 \wedge INIT \wedge SELECT \wedge \neg EVAL \wedge \neg CROSS \wedge \neg MUTATE \rangle$
progress
$\neg$ **FP** $\wedge gen = x \mapsto gen = x + 1$

All that is required to map the design to a sequential architecture is a specific ordering for all assignment statements (a control structure). At the top level, a sequence similar to the one shown back in Figure 2.5 is sufficient. Mapping the lower level designs requires the use of looping control structures and sequences of assignments to implement the quantified and enumerated assignment statements respectively (7:24).

Mappings to parallel architectures require additional design decisions. Most important are the parallel decomposition technique and the granularity of the architecture. However, our high-level design is flexible enough to be mapped to any combination of choices.

A coarse-grained, data-decomposition mapping implements an island model parallel genetic algorithm. For this mapping, regular portions of the assignment statements that are quantified by the population size are allocated to processors so that each processor performs all the assignment operations on a subset of the global population. Internally, each processor contains a serial mapping. Any communication of solutions between processors can be modeled here as a change in the mapping of sets of assignment statements to different processors.

The UNITY design can also be modeled using a master/slave mapping to implement control decomposition. This approach has been considered in cases where the fitness evaluations take significantly longer than the application of the genetic operators. In this configuration, the genetic operator assignment statements might be mapped to the master processor while the statements that evaluate fitness are distributed to the slaves for computation in parallel. An additional design

```
Program SGA
declare
    type SGA_string is record {
        genes  : array[STRING_LENGTH] of 0..1
        fitness: real }
    type Population is list of SGA_string
    old_pop    : Population
    new_pop    : Population
    gen        : integer
initially
    INIT       = false
    EVAL       = false
    CROSS      = true
    MUTATE     = true
    SELECT     = false
    gen        = 0
assign
    ⟨INIT := true     /*Generate an initial population*/
        if (¬INIT)⟩
    ||
    ⟨EVAL := true [] old_pop := new_pop     /* Evaluate the population*/
        if (0 ≤ gen ≤ MAX_GENS ∧ INIT ∧ CROSS ∧ MUTATE)⟩
    ||
    ⟨gen, SELECT, CROSS, MUTATE, EVAL :=
        gen + 1, true, false, false, false     /*Select a new population*/
            if(0 ≤ gen ≤ MAX_GENS ∧ EVAL ∧ ¬SELECT)⟩
    ||
    ⟨CROSS := true     /*Perform crossover*/
        if (0 ≤ gen ≤ MAX_GENS ∧ SELECT ∧ ¬CROSS)⟩
    ||
    ⟨MUTATE := true     /*Perform mutation*/
        if (0 ≤ gen ≤ MAX_GENS ∧ SELECT ∧ ¬MUTATE)⟩
end
```

Figure 4.1 UNITY Description of a Simple Genetic Algorithm

```
Function Initialize
assign
    ⟨||i : 0 ≤ i < POP_SIZE ::     /*For each member of the population*/
        ⟨||j : 0 ≤ j < STRING_LENGTH ::     /*Initialize every bit of the string*/
            new_pop[i].genes[j] := Randint(0, 1)⟩⟩     /*to a random value*/
end
```

Figure 4.2 UNITY Description of SGA Initialization

Function *Select*
always
$\langle || i : 0 \leq i < POP\_SIZE ::$
$cumulative[i] := \langle + j : 0 \leq j \leq i :: old\_pop[j].fitness \rangle \rangle$ /*Relative selection prob. accumulator*/
assign
$\langle || i : 0 \leq i < POP\_SIZE ::$
$new\_pop[i] := old\_pop[j]$ /*Roulette wheel selection*/
if $(cumulative[j-1] < \text{Random}(0, cumulative[POP\_SIZE - 1]) \leq cumulative[j]) \rangle$
end

Figure 4.3 UNITY Description of Roulette-Wheel Selection

Function *Cross*
assign
$\langle || i : 0 \leq i < POP\_SIZE \wedge \text{even}(i) ::$
$\langle \exists j : j = \text{Randint}(0, STRING\_LENGTH - 2) ::$ /*Selected crossover point*/
$new\_pop[i].genes, new\_pop[i+1].genes :=$ /*Create 2 children from*/
/*Parent #1 head and Parent #2 tail*/
$\text{concat}(new\_pop[i].genes[0..j], new\_pop[i+1].genes[j+1..STRING\_LENGTH - 1])$,
/*Parent #2 head and Parent #1 tail*/
$\text{concat}(new\_pop[i+1].genes[0..j], new\_pop[i].genes[j+1..STRING\_LENGTH - 1])$
if $(\text{Random}(0,1) < CROSSOVER\_PROBABILITY) \rangle \rangle$
end

Figure 4.4 UNITY Description of Single Point Crossover

Function *Mutate*
assign
$\langle || i : 0 \leq i < POP\_SIZE ::$
$\langle || j : 0 \leq j < STRING\_LENGTH ::$
$new\_pop[i].genes[j] := (new\_pop[i].genes[j] + 1) \bmod 2$ /*Change a bit*/
if $(\text{Random}(0,1) < MUTATION\_PROBABILITY) \rangle \rangle$
end

Figure 4.5 UNITY Description of Bitwise Mutation

decision determines whether complete evaluations are assigned to processors or component terms are distributed.

4.1.2 Messy Genetic Algorithm. Figure 4.6 reflects the top-level design of what a messy genetic algorithm should accomplish. Detailed descriptions of what the individual components do are shown in Figures 4.7 through 4.11 for PEI, tournament selection, cut, and splice. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. PEI is also $\mathcal{O}(1)$ given enough processors due to the exclusive use of parallel bars (||). Tournament selection and cut-and-splice however, are $\mathcal{O}(POP\_SIZE)$ because the specification calls for selection without replacement and constant population size during cut-and-splice. Given that population size has been shown to be proportional to string length (35), we conclude that these two operators are $\mathcal{O}(l)$.

The following properties, similar to the simple GA's, are evident from the mGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the second assignment statement. The fixed point states that MAX_GENS generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

invariant
$0 \leq gen \leq MAX\_GENS + 1$
FP $\equiv$
$\langle gen = MAX\_GENS + 1 \wedge INIT \wedge TOURNAMENT \wedge \neg CUT\_AND\_SPLICE \rangle$
progress
$\neg$ **FP** $\wedge gen = x \mapsto gen = x + 1$

Mappings to serial and parallel architectures based on decomposition techniques and granularity are similar to SGA mappings. A serial mapping would result in an algorithm that resembles the one shown before in Figure 2.6. It's also possible to map PEI, the primordial phase, and the juxtapositional phase independently so that each would be implemented differently. This piecewise mapping creates more design choices for data distribution and communication strategies for mGAs.

4.1.3 Fast Messy Genetic Algorithm. Figure 4.12 reflects the top-level design of what a fast messy genetic algorithm should accomplish. Detailed descriptions of what PCI and tournament selection do are shown in Figures 4.13 and 4.14. Cut-and-splice remains the same as in a messy

Program *MGA*
declare
type *MGA_string* is record {
alleles : array[$STRING\_LENGTH \cdot EXTENSION$] of integer range $CARDINALITY$
loci : array[$STRING\_LENGTH \cdot EXTENSION$] of integer range $STRING\_LENGTH$
fitness : real }
type *Population* is list of *MGA_string*
old_pop : *Population*
new_pop : *Population*
always
$C = CARDINALITY$
$k = BLOCK\_SIZE$
$l = STRING\_LENGTH$
$POP\_SIZE = C^k \binom{l}{k}$
initially
$INIT$ = false
CUT_AND_SPLICE = false
$TOURNAMENT$ = false
gen = 0
curr_pop_size = POP_SIZE
assign
$\langle INIT :=$ true /*Generate all building blocks—PEI*/
if $(\neg INIT)\rangle$
||
$\langle TOURNAMENT, gen := (gen = PRIMORDIAL\_GENS), gen + 1$ /*Primordial*/
if $(0 \leq gen \leq PRIMORDIAL\_GENS \wedge INIT \wedge \neg TOURNAMENT)\rangle$ /*Phase*/
||
$\langle CUT\_AND\_SPLICE, TOURNAMENT :=$ true, false
if $(PRIMORDIAL\_GENS < gen \leq MAX\_GENS \wedge$ /*Juxtapositional*/
$TOURNAMENT \wedge \neg CUT\_AND\_SPLICE)\rangle$
||
$\langle TOURNAMENT, CUT\_AND\_SPLICE, gen :=$ true, false, $gen + 1$
if $(PRIMORDIAL\_GENS < gen \leq MAX\_GENS \wedge$ /*Phase*/
$CUT\_AND\_SPLICE \wedge \neg TOURNAMENT)\rangle$
end

Figure 4.6 UNITY Description of a Messy Genetic Algorithm

Function PEI
declare
type BB is array[k] of integer range C
$building\_blocks$: array[C^k] of BB
always
$\langle || i : 0 \leq i < C^k ::$
$\langle || j : 0 \leq j < k ::$
$building\_blocks[i][j] = i \bmod j \rangle \rangle$ /*All C^k values for k-size building blocks*/
$combinations$ = list $x | x \in \mathcal{P}(\{0, 1, ..., l-1\}) \wedge |x| = k$ /*l choose k combinations*/
assign
$\langle || i : 0 \leq i < \binom{l}{k} ::$
$\langle || j : 0 \leq j < C^k ::$
$new\_pop[(i \cdot C^k) + j].loci, new\_pop[(i \cdot C^k) + j].alleles :=$
$combinations[i], building\_blocks[j] \rangle \rangle$ /*Create initial population*/
[]
$\langle EVAL :=$ true [] $old\_pop := new\_pop \rangle$ /*Evaluate initial population*/
end

Figure 4.7 UNITY Description of Partially Enumerative Initialization

GA. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. PCI and the primordial phase are now $\mathcal{O}(l)$ because the building blocks can be generated from a much smaller initial population using tournament selection and building block filtering. Tournament selection and cut-and-splice are still $\mathcal{O}(POP\_SIZE)$ for the same reasons as their messy GA counterparts. After applying the $POP\_SIZE \propto l$ transformation, all operators are $\mathcal{O}(l)$.

The following properties, similar to the mGA's, are evident from the fmGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the second assignment statement. The fixed point states that MAX_GENS generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

invariant
$0 \leq gen \leq MAX\_GENS + 1$
FP $\equiv$
$\langle gen = MAX\_GENS + 1 \wedge INIT \wedge TOURNAMENT \wedge \neg CUT\_AND\_SPLICE \rangle$
progress
$\neg$ **FP** $\wedge gen = x \mapsto gen = x + 1$

Function *Tournament_Selection*
always
$\lambda_i = |old\_pop[i].alleles|$
$\theta_{ij} = \left\lceil \frac{\lambda_i \lambda_j}{l} \right\rceil$
$compatible_{ij} = |cand_i.loci \cap cand_j.loci| \geq \theta_{ij}$
$comp_i = (\langle \min\ j : i < j \leq i + n_{sh} \wedge i < j < curr\_pop\_size \wedge compatible_{ij} = \text{true} :: j \rangle)$
initially
$curr = 0$
$new\_index = 0$
$sequencer = 0$
$n_{sh} = SHUFFLES$
assign
$\langle curr\_pop\_size := \frac{curr\_pop\_size}{REDUCTION\_FACTOR}$ /*Reduce the population size*/
if $(0 < gen \leq PRIMORDIAL\_GENS \wedge gen \bmod REDUCTION\_RATE = 0)\rangle$
[]
$\langle$ $\langle []i : 0 \leq i < curr\_pop\_size ::$
$\langle \exists j : j = \text{Randint}(i, curr\_pop\_size - 1) ::$
$old\_pop[i], old\_pop[j], sequencer := old\_pop[j], old\_pop[i], 1$
if $(sequencer = 0)\rangle\rangle$ /*Permute the population order*/
||
$\langle$ $\langle new\_pop[new\_index], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index, curr :=$
$old\_pop[curr], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$
/*Choose a string into the next generation based on its fitness and length*/
if $(comp_{curr} \leq curr + n_{sh} \wedge$
$(old\_pop[curr].fitness$ betterthan $old\_pop[comp_{curr}].fitness \vee$
$(old\_pop[curr].fitness = old\_pop[comp_{curr}].fitness \wedge \lambda_{curr} < \lambda_{comp_{curr}}))) \sim$
$old\_pop[comp_{curr}], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$
if $(comp_{curr} \leq curr + n_{sh} \wedge$
$(old\_pop[comp_{curr}].fitness$ betterthan $old\_pop[curr].fitness \vee$
$(old\_pop[comp_{curr}].fitness = old\_pop[curr].fitness \wedge \lambda_{comp_{curr}} < \lambda_{curr}))) \sim$
$old\_pop[curr], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index + 1, curr + 1$
if $(comp_{curr} > curr + n_{sh})\rangle$
if $(new\_index < curr\_pop\_size \wedge sequencer = 1)\rangle$
||
$\langle sequencer := 0$ /*Permute population again*/
if $(new\_index < curr\_pop\_size \wedge curr \geq curr\_pop\_size)\rangle$
||
$\langle old\_pop := new\_pop$
if $(new\_index = curr\_pop\_size)\rangle\rangle$
end

Figure 4.8 UNITY Description of mGA Tournament Selection

Function *Cut_and_Splice*
declare
cut_list : list of *MGA_STRING*
initially
new_index = 0
sequencer = 0
curr = 0
CUT = false
SPLICE = true
assign
$\langle [] i : 0 \leq i < curr\_pop\_size ::$
$\langle \exists j : j = \text{Randint}(i, curr\_pop\_size - 1) ::$
$old\_pop[i], old\_pop[j], sequencer := old\_pop[j], old\_pop[i], 1$
if $(sequencer = 0)\rangle\rangle$ /*Permute the population order*/
||
$\langle CUT, SPLICE, curr, sequencer :=$
true, false,*curr, sequencer* /*Perform a cut operation*/
if $(\neg CUT \wedge SPLICE \wedge curr < POP\_SIZE \wedge sequencer = 1) \sim$
false, true,$curr + 2, sequencer$ /*Perform a splice operation*/
if $(CUT \wedge \neg SPLICE \wedge curr < POP\_SIZE \wedge sequencer = 1) \sim$
$CUT, SPLICE, 0, 0$ /*Permute population again*/
if $(curr \geq POP\_SIZE)\rangle$
end

Figure 4.9 UNITY Description of Cut and Splice

Function *Splice*
initially
$cut\_index = head$
assign
$\langle new\_population[new\_index].allele, new\_population[new\_index].loci, cut\_index :=$
/*Copy single string if only one string left or splice probability not met*/
$cut\_list[cut\_index].allele, cut\_list[cut\_index].loci, cut\_index + 1$
if $(cut\_index = tail - 1 \vee$ Random(0,1)$> P_s) \sim$
/*Splice two strings together otherwise*/
concat$(cut_list[cut_index].allele, cut_list[cut_index + 1].allele)$,
concat$(cut_list[cut_index].loci, cut_list[cut_index + 1].loci), cut_index + 2$
otherwise$\rangle$
end

Figure 4.10 UNITY Description of Splice

Function *Cut*
always
$\lambda_i = |old\_pop[i].alleles|$
assign
$\langle$ $\langle \exists j : j = \text{Randint}(0, \lambda_i - 2) ::$ /*Cut first mate & save pieces, based on cut probability*/
$cut\_list[head].loci, cut\_list[head].alleles, cut\_list[tail].loci, cut\_list[tail].alleles :=$
$old\_pop[i].loci[0..j], old\_pop[i].alleles[0..j],$
$old\_pop[i].loci[j+1..\lambda_i - 1], old\_pop[i].alleles[j+1..\lambda_i - 1]$
if (Random(0,1) $< P_c\lambda_i$) $\sim$
$old\_pop[i].loci, old\_pop[i].alleles, null, null$
otherwise$\rangle$
||
$\langle \exists j : j = \text{Randint}(0, \lambda_{i+1} - 2) ::$ /*Cut second mate & save pieces, based on cut probability*/
$cut\_list[tail-1].loci, cut\_list[tail-1].alleles, cut\_list[head+1].loci, cut\_list[head+1].alleles :=$
$old\_pop[i+1].loci[0..j], old\_pop[i+1].alleles[0..j],$
$old\_pop[i+1].loci[j+1..\lambda_{i+1} - 1], old\_pop[i+1].alleles[j+1..\lambda_{i+1} - 1]$
if (Random(0,1) $< P_c\lambda_{i+1}$) $\sim$
$old\_pop[i+1].loci, old\_pop[i+1].alleles, null, null$
otherwise$\rangle\rangle$
[]
$\langle total\_strings :=$ /*Tally the final number of string segments*/
4 if $(cut\_list[head+1].loci = cut\_list[tail].loci = null)$ $\sim$
2 if $(cut\_list[head+1].loci \neq null \wedge cut\_array[tail].loci \neq null)$ $\sim$
3 otherwise$\rangle$
end

Figure 4.11 UNITY Description of Cut

Program $FMGA$
declare
type MGA_string is record {
$alleles$: array[$STRING\_LENGTH \cdot EXTENSION$] of integer range $CARDINALITY$
$loci$: array[$STRING\_LENGTH \cdot EXTENSION$] of integer range $STRING\_LENGTH$
$fitness$: real }
type $BBF\_schedule$ is record {
gen : integer
λ : integer
θ : integer }
type $Population$ is list of MGA_string
old_pop : $Population$
new_pop : $Population$
always
$C = CARDINALITY$
$k = BLOCK\_SIZE$
$l = STRING\_LENGTH$
$n_a = N_a$
$n_g = \left(\frac{l}{l-k}\right)^k$
$pop\_size = n_a \cdot n_g$
initially
$INIT$ = false
CUT_AND_SPLICE = false
$TOURNAMENT$ = false
gen = 0
assign
$\langle INIT := \text{true} \quad \text{if } (\neg INIT)\rangle$ /*Generate initial population*/
||
$\langle TOURNAMENT, gen := (gen = PRIMORDIAL\_GENS), gen + 1$ /*Primordial*/
if $(0 \leq gen \leq PRIMORDIAL\_GENS \wedge INIT \wedge \neg TOURNAMENT)\rangle$ /*Phase*/
||
$\langle CUT\_AND\_SPLICE, TOURNAMENT :=$ true, false
if $(PRIMORDIAL\_GENS < gen \leq MAX\_GENS \wedge$ /*Juxtapositional*/
$TOURNAMENT \wedge \neg CUT\_AND\_SPLICE)\rangle$
||
$\langle TOURNAMENT, CUT\_AND\_SPLICE, gen :=$ true, false, $gen + 1$
if $(PRIMORDIAL\_GENS < gen \leq MAX\_GENS \wedge$ /*Phase*/
$CUT\_AND\_SPLICE \wedge \neq TOURNAMENT)\rangle$
end

Figure 4.12 UNITY Description of a Fast Messy Genetic Algorithm

```
Function PCI
declare
    loci_array : array[l] of integer
initially
    ⟨||i : 0 ≤ i < l ::
        loci_array[i] = i⟩
    ⟨||i : 0 ≤ i < POP_SIZE ::
        GENERATED_i = false⟩
assign
    ⟨||i : 0 ≤ i < pop_size ::
        ⟨[]j : 0 ≤ j < l − k ::
            ⟨∃x : x = Randint(j, l − 1) ::     /*Randomly generate loci and alleles*/
                new_pop[i].loci[j], new_pop[i].alleles[j], loci_array[j], loci_array[x], GENERATED_i :=
                    loci_array[x],Randint(0, C − 1), loci_array[x], loci_array[j], true
                        if (¬GENERATED_i)⟩⟩
        ||
        ⟨EVAL := true [] old_pop := new_pop     /*Evaluate the new population*/
            if (⟨∧i : 0 ≤ i < POP_SIZE :: GENERATED_i⟩)⟩
end
```

Figure 4.13 UNITY Description of Probabilistically Complete Initialization

Mappings to serial and parallel architectures based on decomposition technique and granularity are similar to mGA mappings. A serial mapping would result in an algorithm that resembles the one shown in Figure 2.7. PCI, the primordial phase, and the juxtapositional phase can again be independently mapped so that each would be implemented differently. The same options for data distribution and communication strategies apply from mGAs.

4.2 GA Low-Level Design and Implementation

Specific data and control structures are the items to be addressed during low-level design and implementation. The following sections describe these major components of each genetic algorithm. In each case, the flexibility of the target language ('C') has been used to full advantage. This is especially evident in the initial use of pointers and dynamic memory allocation to accommodate run-time specification of data structure sizes, but subsequent access of those data structures using array notation.

4.2.1 Simple Genetic Algorithm. The control structure used by Genesis is a direct instantiation of the algorithm presented in Figure 2.5. The high-level design (Figure 4.1) is mapped

Function *Tournament_Selection*
always

$\lambda_i = |old\_pop[i].alleles|$

$\theta_{ij} = \left\lceil \frac{\lambda_i \lambda_j}{l} \right\rceil$

$compatible_{ij} = |cand_i.loci \cap cand_j.loci| \geq \theta_{ij}$

$comp_i = (\langle \min\ j : i < j \leq i + n_{sh} \wedge i < j < pop\_size \wedge compatible_{ij} = \text{true} :: j \rangle)$

initially

$curr = 0$

$new\_index = 0$

$sequencer = 0$

$n_{sh} = SHUFFLES$

$next\_filter = 1$

assign

$\langle$ $\langle || i : 0 \leq i < pop\_size ::$
$\langle [] j : 0 \leq j < schedule[next\_filter].\lambda - schedule[next\_filter - 1].\lambda ::$
$\langle \exists x : x = \text{Randint}(0, \lambda_i) ::$
$\langle || y : x < y < \lambda_i ::$ /*Perform BBF*/
$old\_pop[i].loci[y-1], old\_pop[i].alleles[y-1] :=$
$old\_pop[i].loci[y], old\_pop[i].alleles[y] \rangle \rangle \rangle \rangle$
if $(gen = schedule[next\_filter].gen) \rangle$

$[]$

$\langle$ $\langle [] i : 0 \leq i < pop\_size ::$
$\langle \exists j : j = \text{Randint}(i, pop\_size - 1) ::$ /*Permute the population order*/
$old\_pop[i], old\_pop[j], sequencer := old\_pop[j], old\_pop[i], 1$
if $(sequencer = 0) \rangle \rangle$

||

$\langle$ $\langle new\_pop[new\_index], old\_pop[curr+1], old\_pop[comp_{curr}], new\_index, curr :=$
$old\_pop[curr], old\_pop[comp_{curr}], old\_pop[curr+1], new\_index + 1, curr + 2$
/*Choose a string into the next generation based on its fitness and length*/
if $(comp_{curr} \leq curr + n_{sh} \wedge$
$(old\_pop[curr].fitness$ betterthan $old\_pop[comp_{curr}].fitness \vee$
$(old\_pop[curr].fitness = old\_pop[comp_{curr}].fitness \wedge \lambda_{curr} < \lambda_{comp_{curr}}))) \sim$
$old\_pop[comp_{curr}], old\_pop[comp_{curr}], old\_pop[curr+1], new\_index + 1, curr + 2$
if $(comp_i \leq curr + n_{sh} \wedge$
$(old\_pop[comp_{curr}].fitness$ betterthan $old\_pop[curr].fitness \vee$
$(old\_pop[comp_{curr}].fitness = old\_pop[curr].fitness \wedge \lambda_{comp_{curr}} < \lambda_{curr}))) \sim$
$old\_pop[curr], old\_pop[curr+1], old\_pop[comp_{curr}], new\_index + 1, curr + 1$
if $(comp_{curr} > curr + n_{sh}) \rangle$
if $(new\_index < pop\_size \wedge sequencer = 1) \rangle$

||

$\langle sequencer := 0$ if $(new\_index < pop\_size \wedge curr \geq pop\_size) \rangle$ /*Permute population again*/

||

$\langle old\_pop := new\_pop$ if $(new\_index = pop\_size) \rangle \rangle$

end

Figure 4.14 UNITY Description of fmGA Tournament Selection

to a sequence of statement enclosed inside a loop from 0 to the maximum number of generations. The string data structure contains both the genes and a location to store the fitness.

```
typedef struct {
   char *Gene;
   double Perf;
   int Needs_evaluation;
} STRUCTURE;
```

Two populations are dynamically allocated during initialization, one to hold the current population and one accept the new strings created by the recombination operators. The dual access modes that 'C' provides enable the programmer to optimize specific operations. Direct access to individuals in the population is available through the array-indexing mode. A newly created population can be moved to the old population by swapping the pointers to each population. One of these two operations would be significantly slower if only one access mode were available.

4.2.2 Messy and Fast Messy Genetic Algorithms. The control structures for the mGA and fmGA are direct instantiations of their respective algorithms presented in Figures 2.6 and 2.7. Their high-level designs (Figures 4.6 and 4.12) are mapped in the same fashion as the simple GA. The string data structure is modified to store the additional locus information.

```
typedef struct {
   int    *locus;
   char   *allele;
   double fitness;
} mgastring;
```

The population data structures are implemented exactly like the simple GA population with all the associated benefits.

4.3 Genetic Algorithm Fitness Functions for Energy Minimization

A major objective in any research effort is to be able to compare results to previous research. In order to make these comparisons, the current effort must provide accurate results and the relationships between compared works must be understood. This section discusses the accuracy of our energy model/fitness function. Appendix C contains details on the relationship between data files and procedures for comparing results obtained from the two different energy models used in previous research (68, 3).

The design and implementation of a fitness function is critical to the successful application of genetic algorithms to a problem domain. For most complex problems of interest, the majority of the total execution time is spent evaluating the quality of solutions. Thus it is imperative that the detailed design and implementation are as efficient as possible. In the research environment however, some inefficiency may be tolerated to take advantage of other software characteristics. In particular, we prefer to remain isolated from machine dependent math libraries in the interest of enhancing portability.

4.3.1 Previous Energy Model Designs. Two energy model implementations are available for our use with genetic algorithms: ECEPP and CHARMm. A description of the ECEPP design and implementation is given in (21). There are several drawbacks to using ECEPP as a general energy model. The greatest hindrance is the limited size (≈ 50 amino acids) and types of proteins that it can accommodate (21). The configuration management complexity of this implementation is also very high because we have to interface C and Fortran code, and portability is poor because ECEPP requires special math libraries that aren't widely available on all AFIT computer systems.

The design specification for CHARMm is given in Brooks (4), however the only implementation is proprietary (QUANTA (8)) so AFIT designed and implemented its own software for use with genetic algorithms (3). Because it's the only force field energy model capable of modeling general organic molecules, the CHARMm energy model is essential to the Air Force's research of non-linear optical (NLO) materials. The design specifies list and array data structures that are suitably efficient for the highly sequential, read-only access required by the majority of the program. Our design is: scalable over all protein sizes (within machine limits) and types, written in C for simple integration with our GAs, and requires nothing more than standard C header files.

ECEPP and CHARMm are both force-field energy models so we expect their time complexity to be $\mathcal{O}(n^2)$, where n is the number of atoms in the protein, because force-field models account for all pairwise interactions between atoms (57, 58). Our CHARMm implementation uses lists of the pairwise interactions and constant time calculations of the energy components to realize the minimum time complexity of the design. The ECEPP implementation is also $\mathcal{O}(n^2)$ (21).

Table 4.3 Energy Component Comparison

Implementation Component	Previous Implementation (kcal/mol)	QUANTA Values (kcal/mol)
Bond (ρ)	12.380	12.374
Bond Angle (θ)	6.189	6.183
Dihedral Angle (ϕ)	202.479	8.204
Lennard-Jones	-45.133	-15.014
Electrostatic	-0.267	-40.973
Total	175.649	-29.225

4.3.2 Energy Model Design Enhancements. The model created by (3) was roughly 90% accurate as an implementation of the CHARMm energy minimization function and data structures. Table 4.3 compares the individual energy components from this model with the values obtained from QUANTA for a specific conformation of [Met]-enkephalin. The design enhancements discussed in the following paragraphs have been implemented to eliminate the discrepancies between the two implementations.

Since the bond lengths and bond angles are being held fixed, we'd expect no differences between the two CHARMm implementations for those energy terms. The differences shown are less than one tenth of one percent and are at least three orders of magnitude smaller than the relative error of the other terms. The dihedral angle energy is the greatest source of error, followed by the Lennard-Jones potential and electrostatic terms. A technical review of this CHARMm energy model design and implementation uncovered several deficiencies that have been resolved as part of this effort.

1. There are actually three classes of dihedral angles—fixed, independent, and *dependent.* Dependent dihedral angles weren't handled in the original design. Any rotation of an independent dihedral angle without a corresponding rotation of its dependent dihedral angles (Figure 4.15) results in a higher energy value for the system. Figure 4.16 illustrates how the molecules should react to a rotation of the independent dihedral angle.
2. QUANTA treats the two atoms at the ends of a group defining a dihedral angle as a special case, non-bonded interaction. The energy contribution of this *1-4 non-bonded interaction* is calculated as one-half of the normal non-bonded energy value between non-bonded atom pairs.
3. Dihedral angles were encoded assuming a range of $-\pi$ to π, however they were then mistakenly decoded in the range of 0 to 2π.

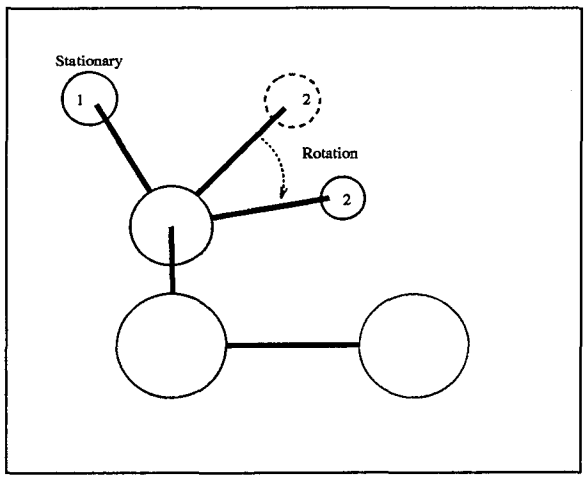


Figure 4.15 Incorrect Dependent Dihedral Angle Rotation: Dependent atom (1) remains stationary while independent atom (2) rotates.

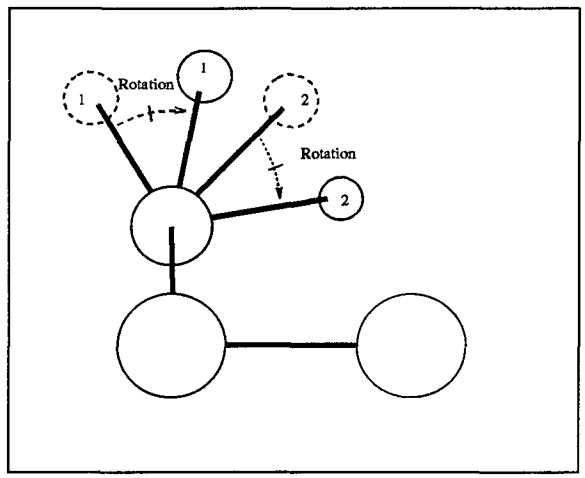


Figure 4.16 Correct Dependent Dihedral Angle Rotation: Dependent atom (1) rotates in unison with the rotation of independent atom (2).

4. The dihedral energy term should include a parameter for the *periodicity* of the angle. Together, the periodicity (n) and γ (see Equation 3.1) define the number of minimum energy dihedral angles and their values (see Figure 4.17). Only periodicities of n = 1, 2, 3, 4, and 6 are allowed, although a single dihedral angle might have more than one periodicity associated with it.
5. A units conversion factor of $\approx$ 332.0 was missing from the electrostatic energy calculation. This factor represents a relative dielectric constant of one ($\epsilon_{rel} = 1$).

4.3.3 Implementation Details. Our C implementation of the CHARMm energy model reflects the previously noted priority of portability over efficiency during this research phase. While highly optimized matrix-matrix and matrix-vector operations are available in various math libraries (50, 53), they weren't incorporated. Instead C versions were written to enhance machine portability. This has paid dividends in the ease with which both the GA and energy function code has been ported from SPARCstations to Silicon Graphics and IBM PCs as well as the iPSC/2, iPSC/860, and Paragon parallel computing platforms.

Certain minor inefficiencies have been introduced by the modifications required to validate the energy model. These inefficiencies are either one-time costs incurred during data structure initialization or a small number of recurring costs during the energy calculation. In either case, the code is documented to identify the source of the inefficiency and any known possible solutions to

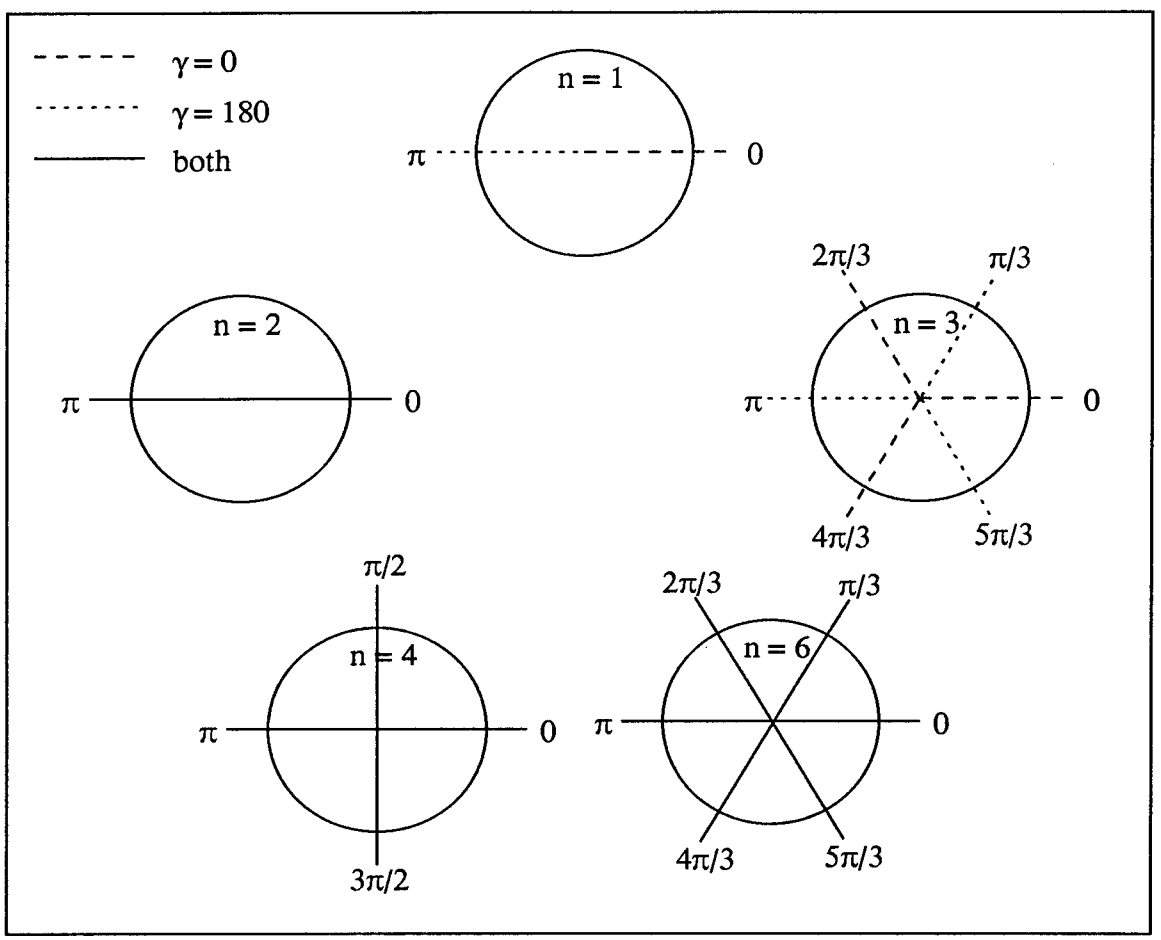


Figure 4.17 Dihedral Angle Periodicity

the problem. No modifications have resulted in an increase in the overall time complexity of the energy function which has been shown consistent with the $\mathcal{O}(n^2)$ prediction from the design (31).

Item 1 on the deficiencies list is the most difficult modification to implement. It requires the addition of two fields to the $ATOM\_TYPE$ data structure and additional initialization code. Each dihedral angle must identify an independent dihedral angle that it's dependent on and the angular value of that relationship. Correcting this item won't change any of the energy terms shown in Table 4.3, however, this change is critical to correctly calculating the coordinates of atoms in the protein due to rotations of the independent dihedral angles during GA processing. Without this change, the genetic algorithm would attempt to minimize the difference between population members and the input rather than optimize the conformation!

Item 2 requires an additional pairwise interaction list to process the 1-4 non-bonded interactions. The creation of this list can be easily inserted into the section that generates the other interaction lists and processing the list requires only one additional call to the function that calculates non-bonded interaction energies. The only difference between these non-bonded interactions and all the others on the NON_BONDED list is the fact that the results for 1-4 non-bonded interactions are scaled by one half. This modification will change the values returned for the electrostatic and Lennard-Jones energy terms.

Items 3, 4 and 5 from the deficiencies list are simple, one-line changes represented by transformations 4.1 through 4.3.

$$dihedral \cdot (1 + \cos(dihed\_angle - \gamma)) \Rightarrow dihedral \cdot (1 + \cos((n \cdot dihed\_angle) - \gamma)) \quad (4.1)$$

$$atom[i].dihed\_angle = \left(\frac{(360\pi)temp}{180 \cdot 2^{slice}}\right) \Rightarrow atom[i].dihed\_angle = \left(\frac{(2\pi)temp}{2^{slice}}\right) - \pi \quad (4.2)$$

$$\frac{(atom[i].chg)(atom[j].chg)}{r} \Rightarrow \frac{(atom[i].chg)(atom[j].chg)prop\_const}{r} \quad (4.3)$$

The first two changes should correct the dihedral energy term. If the last modification were made in isolation the electrostatic energy term would still be incorrect ($-0.267 \times 332 = -88.644 \not\approx -40.973$). This change combined with the resolution of deficiency 2 should correct both the electrostatic and Lennard-Jones potential.

The energy values obtained from the original energy model didn't exhibit any correlation with the values obtained from QUANTA for identical protein structures. A strong correlation between the two implementations is the goal we seek to obtain with these energy model enhancements.

4.4 Summary

Software reuse is a major design decision for genetic algorithm implementations at AFIT. Effort is expended on design recapture using UNITY to aid in the understanding of that software. The designs presented in this chapter have been created using a top-down design methodology based on the existing 'C' code. The performance of computer programs is judged by their efficiency and effectiveness. This chapter also developed and compared the calculated efficiencies of implementations with the design-predicted efficiencies for the major components of three genetic algorithms. The effectiveness of the CHARMm energy implementation is examined and a plan to improve its accuracy is designed. The next chapter describes the test setup to measure these performance characteristics.

V. *Genetic Algorithm Experiment Designs*

While examples can't be used to prove anything, experiments serve a very important purpose in many areas of investigation! An experiment may provide support for our belief in a hypothesis and/or a better understanding of the theory behind the conjecture. Used correctly, experiments are also effective tools for disproving theories (counter-examples) and suggesting alternate hypotheses. The experiments in this investigation are designed to be of the support type. The results are used to identify areas of the problem domain that require additional research.

Section 5.1 identifies the specific protein used as a bench-mark test. Section 5.2 defines the process used to validate the effects of energy model enhancements. Sections 5.3 and 5.4 describe experiments that use the test molecule and enchanced energy model to evaluate simple GA parameter sets and compare the performance of several genetic algorithms.

5.1 Test Molecule

Figure 5.1 is a representation of an extended conformation of our test molecule, [Met]-enkephalin. [Met]-enkephalin is a very small pentapeptide defined by the five-amino-acid sequence *Tyr-Gly-Gly-Phe-Met.* This molecule was chosen because its native conformation is known and other researchers have attempted to predict its tertiary structure using energy minimization (68, 85, 57, 3). Using neutral NH_2 and -COOH groups as terminators at the α-amino and α-carboxyl ends respectively, there are 24 independently variable dihedral angles that influence the tertiary structure of [Met]-enkephalin. These dihedral angles are labeled along their center bonds in Figure 5.1. Table 5.1 shows the values of these dihedrals for the accepted energy minimum. The energy values in ECEPP and QUANTA for this conformation are -14.593 and -29.225 kcal/mol respectively.

5.2 Energy Model Validation

Validation of our energy model consists of comparing the energy values produced by our implementation with those from QUANTA (see Appendix C for a description of the comparison process). Although no known verification of their model has been accomplished, this investigation assumes QUANTA's implementation is correct. Three conformations of [Met]-enkephalin are used

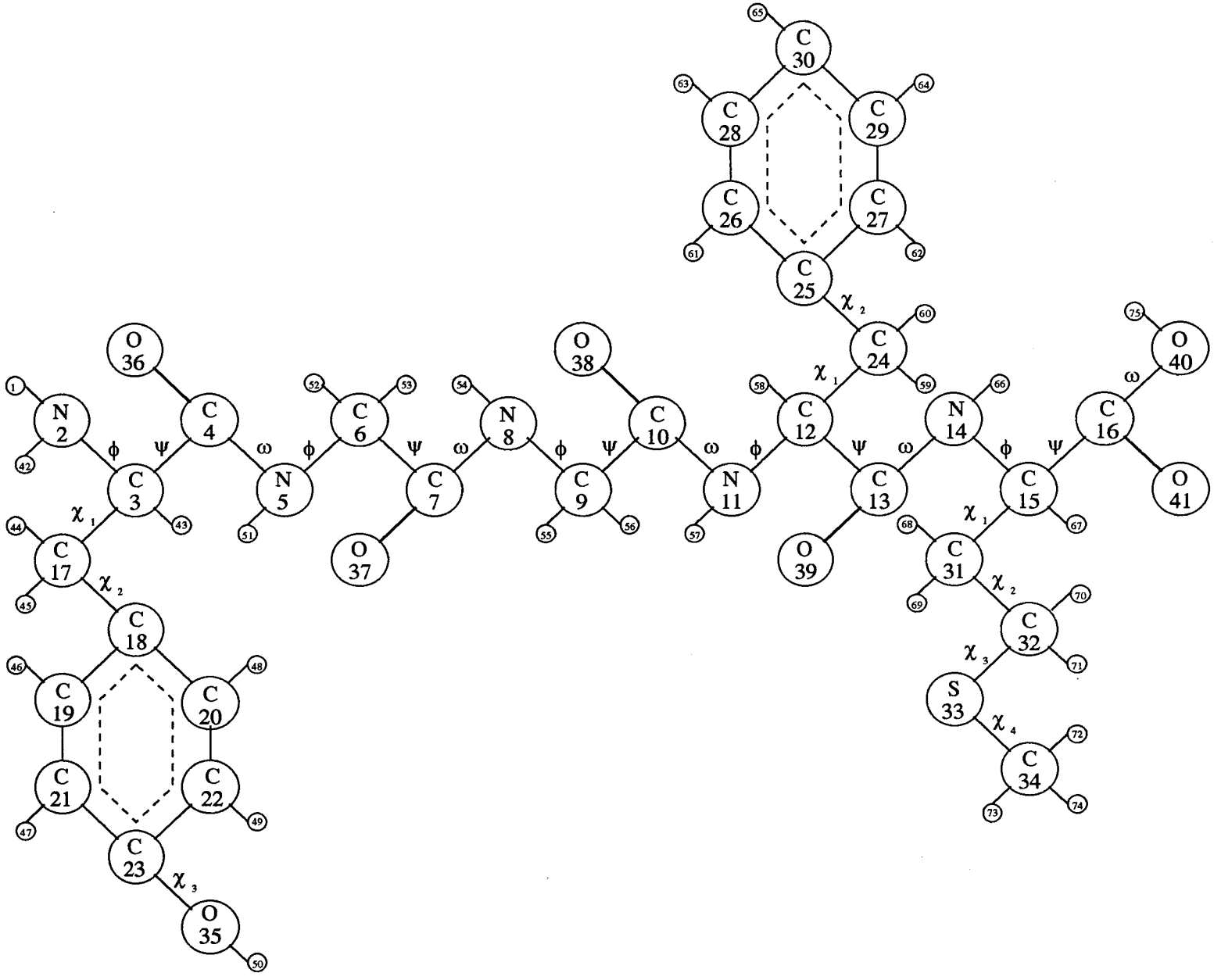


Figure 5.1 Extended Conformation of [Met]-enkephalin

Table 5.1 Dihedral Angles for Accepted Energy Minimum [Met]-enkephalin

	Dihedral Angle (degrees)						
Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr	-86	156	-177	-173	79	166	
Gly	-154	83	169				
Gly	84	-74	170				
Phe	-137	19	-174	59	-85		
Met	-164	160	-180	53	175	-180	-59

as comparison cases: the accepted global energy minimum described in section 5.1 above, a semi-optimal conformation found by a short simple genetic algorithm run, and a randomly generated conformation. These test conformations represent a wide range in the conformational search space. An attempt is made to explain discrepancies discovered during these comparisons.

5.3 Simple Genetic Algorithm Parameters for Protein Energy Minimization

This section describes some preliminary findings using theoretical population sizing formulas and their inapplicability to the problem at hand. These findings provide justification for the repetition of a previous experiment, using our energy minimization function in place of the original functions. The results from these experiments are used to guide parameter set selection for the experiments described in Section 5.4 and to explain some results from this investigation and Brinkman's (3).

5.3.1 Problems with Conservative Theoretical Population Sizing. Goldberg *et.al.* analyze the population sizing problem in (35). The formula they derive is based on conservative assumptions and is therefore an upper bound for choosing a population size. GAs using the calculated population size should converge somewhere between $\mathcal{O}(l \log l)$ and $\mathcal{O}(l^2 \log^3 l)$ function evaluations depending on the selection scheme used. Goldberg's formula, without considering additional terms to account for noisy operators, is

$$n = 2c \left(\frac{\sigma_{rms}^2 (m-1)}{d^2} \right) \chi^k \tag{5.1}$$

where

n is the calculated population size,

Table 5.2 Theoretical Population Size Required for Optimal Solution Convergence of [Met]-enkephalin

Variance Assumption	Calculation Parameters						Population Size
	l	k	χ	c	d	σ^2_{rms}	
Worst Case	240	5	2	6	0.1	1.65×10^{23}	2.98×10^{29}
Measured	240	5	2	6	0.1	9.20×10^{18}	1.66×10^{25}
Best Case	240	5	2	6	0.1	8.95×10^{-48}	1.62×10^{-41}

c is a parameterized constant of the confidence factor you want to enforce (α),

$\sigma^2_{rms}(m-1)$ is an estimate of the variance of the average order-k schema ($m = l/k$),

d is the signal difference we wish to detect,

χ is the cardinality of the encoding alphabet, and

k is the estimated order of deception in the problem.

Table 5.2 summarizes population size values using this model for a genetic algorithm minimizing the energy of [Met]-enkephalin based on the following, conservative assumptions (more realistic assumptions would only *increase* the calculated population size!).

1% sampling error is allowed ($\alpha = 0.01, c \approx 6$),

the signal difference we wish to detect is $d = 0.1$,

the estimated order of deception in the problem is $k = 5$,

the maximum energy is $f_{max} \approx 75^2 \cdot 10^9$,

the minimum energy is $f_{min} \approx 0$, and

three variance estimations:

- Worst case variance—$\sigma^2_{rms} = \frac{(f_{max} - f_{min})^2}{4}$
- Measured variance over 40,000 random conformations—$\sigma^2_{rms} \approx 10^{19}$
- Best case variance—$\sigma^2_{rms} = \frac{(f_{max} - f_{min})^2}{2\chi^l}$

Obviously, the best case calculation doesn't help us because it tells us we don't need a population at all! The other two cases don't provide a useful result either. The calculation time for one conformation of [Met]-enkephalin has been measured at approximately 167 msec at the fastest. At this rate it would take more than one million times the age of the universe just to evaluate the initial population!

No theoretical work to date can provide a useful population size estimate for the simple genetic algorithm approach to protein folding. Also, nobody has even attempted to establish any

Table 5.3 Comparison of Empirically Determined GA Parameter Settings

Author	Population Size	Crossover Rate	Mutation Rate
Schaffer	20 – 30	0.75 – 0.95	0.005 – 0.01
De Jong	50 – 100	0.60	0.00
Grefenstette	30	0.95	0.01

theory behind the choice of good crossover and mutation rates. Therefore, we must currently fall back on empirical methods for the identification of good parameter sets.

5.3.2 Comparison with Other Empirical Results. The most comprehensive empirical analysis of parameter settings was accomplished by Schaffer *et.al.* in (77). They used a set of ten test functions, including De Jong's five-function test suite. The objective of the study was to identify ranges of population size, crossover rates, and mutation rates that would exhibit good online performance over the range of test functions. They also evaluated the effects of one- and two-point crossover and found that the latter was always at least as good as the former. Table 5.3 lists the parameters suggested by Schaffer along with the ones proposed earlier by De Jong (14) and Grefenstette (42).

These results, especially for population size, suggest some alternate interpretations for observations made previously in (3, 65). In both cases, better solution quality was obtained with larger processor counts, however, the global population size was kept constant during these investigations. The better solutions may have been observed because the processor sub-population sizes were reduced to values that foster better GA performance.

5.3.3 Test Design. The following experiments are designed to test the hypothesis that the best parameter settings for our problem are inside the range specified by Schaffer (77:55). With the exception of using two-point crossover exclusively and regular binary encoding instead of gray code, all controls are set exactly as in that study (77:53). A complete factorial design for the 420 remaining parameter combinations is performed, and ten repetitions with different random number seeds are run for each combination.

Since the simple GA has been incapable of finding the accepted global optimum, our definition of "doing well" needs to be different than Schaffer's. We choose the following definition to closely approximate his original: at least 10% (42) of the cells in the design locate a value within 10

kcal/mol of the best known solution (-35.1155 kcal/mol) at least 50% of the time (5 out of 10 repetitions) (77:54). We were also unable to obtain a reference describing the mechanics of the Tukey B test used by Schaffer, so instead we use the Kruskal-Wallis test to identify the members of the *best online pool.* Test cells (parameter combinations) belong to the best online pool if their performance cannot be statistically distinguished from the best performing cell.

5.4 Comparison of SGAs and fmGAs for the Protein Folding Problem

The investigation so far has been mostly preparatory work to enable a comparison of the performance of genetic algorithms while minimizing the energy of [Met]-enkephalin. But first, we must justify the decision to exclude the standard messy GA from our final comparisons. Recall that the initial population size for binary mGAs is $2^k \binom{l}{k}$ where l is the string length and k is the block size. With a block size of $k = 5$, which is a rough estimate for our problem, the initial population size is $2^5 \binom{240}{5} > 2$ billion. This population would require more than 1,000 years to evaluate on a sequential machine and more than four years using the largest parallel machine to which we have access! Thus we must discard messy GAs because they are too computationally expensive to be applied to the protein folding problem.

Now that the scope of our research approach has been defined, we must decide on criteria to judge the performance of the remaining two genetic algorithms. Ultimately, the lowest energy conformation found or the conformation that most nearly resembles the accepted conformation must weigh heavily in our conclusions. However, we must also look at the possibilities for improvement in the algorithms themselves. Simple genetic algorithms have been studied for more than 20 years, and their vulnerability to deception and premature convergence is well-documented even if it isn't fully understood (24, 46, 34, 39). In contrast, messy and fast messy GAs are less than five years old and the papers introducing them are the only theoretical treatments available (39, 37, 38, 36). Forthcoming research by Goldberg and Merkle, as well as this investigation should expose some of the theoretical and practical knowledge required for fmGAs to consistently outperform SGAs!

There are a host of other comparisons that can be made to further distinguish differences between the algorithms if the algorithms are judged nearly equivalent by the primary criteria. We have chosen measures of efficiency for this task for two reasons. First, we want solutions within

an acceptable amount of turn-around time so that the programs have room to scale up to bigger problems. Second, we want to measure how efficiently the resources (processors) we allocate to the problem are used. The quantitative comparisons to be made between the different algorithms include:

- Lowest Average Energy—averaged over 10 independent runs
- Average Execution Time—averaged over 10 independent runs
- Speedup—average serial execution time divided by average parallel execution time ($S = \frac{T_s}{T_p}$)
- Efficiency—speedup divided by the number of processors ($E = \frac{S}{p}$)

5.4.1 Parallel Communication Strategies. AFIT's parallel simple genetic algorithm is implemented on the iPSC/2 and iPSC/860 hypercubes (76). The implementations use four parameters to control communication between processors. One defines the configuration of processors that solutions will be shared with (immigration range): 0 → all other processors, 1 → nearest neighbors in the hypercube, 2 → next processor in node order. The second controls how often solutions are shared: *Epoch* = # of generations between migrations. The third parameter determines how many solutions are sent: 1 → the current best is sent, 2 → the current best is sent if it's better than the solution sent previously, 3 → a percentage of the population is sent. The final parameter specifies the percentage to be shared (immigration rate) if option 2 of the third parameter is chosen.

AFIT's parallel fast messy genetic algorithm is implemented on the Paragon (64). Three communication strategies are available in this implementation. The first option (independent, I) allows each processor to run its fmGA to completion, then each sends its best solution to a node 0 for identification of the overall best. For the second option (combined, C) each processor sends its building blocks to node 0 at the end of the primordial phase, then node 0 processes the global population as a combined juxtapositional phase, while each of the other nodes execute the juxtapositional phase with their smaller subpopulations. The combined strategy ends with node 0 again collecting the best solution from each processor and identifying the best solution found. Finally, the communication strategy implemented for this investigation performs a global exchange of building blocks at the end of the primordial phase (global combine, G). Then every processor executes an independent juxtapositional phase on a copy of the global population.

Table 5.4 Parallel SGA Communication Parameter Settings

Strategy	Immigration Range	Epoch Length	Immigration Rate
Global	All Processors	10 Generations	New Best Only
Neighbor	Hypercube Neighbors	10 Generations	New Best Only
Ring	Next In Order	10 Generations	New Best Only

5.4.2 Test Design. The sequential SGA and fmGA are evaluated as baselines for the performance of the parallel versions. Each algorithm is executed ten times starting with a different random number seed so we can calculate significant statistical averages of the quantitative metrics.

A fully factorial test design for parallel SGAs creates a large number of combinations that's dependent on the number of epoch sizes ($|e|$) and the number of immigration rates ($|i|$) we choose to examine. The number of test cases is given by

$$3 \times 3 \times |e| \times |i|, \tag{5.2}$$

where even small values of $|e|$ and $|i|$ involve more than 50 different combinations. Because it's not the goal of this research to evaluate the performance of these combinations but to evaluate comparable simple and fast messy GAs, we choose a small subset of communication strategies that are similar to the available fmGA strategies. The three communication strategies chosen and their associated parameters are listed in Table 5.4. Two population strategies are also tested. The first keeps the global population size constant (640) so that the number of members in a subpopulation decreases as the number of processors increases. The second strategy fixes the subpopulation size at 20 members so that the global population size grows as the number of processors increases. Ten iterations of each strategy, using different random seeds, are run on hypercubes of dimension 2–5. Figure 5.2 represents a general input file for the parallel SGA. Values separated by commas indicate the values that each parameter may assume in the experimental runs.

We must pause here to contemplate the choice of parameters for the fast messy GA. Since identifying *good* parameters for the fmGA isn't a goal of this effort and there are no treatments of the subject other than in Goldberg's introduction of the algorithm (36), many of the same parameter values are used without modification. The splice probability is set to 1.0 because the operation

```
Population Size          = 640,320,160,80,40
Structure Length         = 240
Crossover Rate           = 0.75
Mutation Rate            = 0.002
Generation Gap           = 1.0
Scaling Window           = 1
Structures Saved/Node    = 2
Max Gens w/o Eval        = 10
Epoch Size               = 10
All soln eval flag       = 0
Solution sharing flag    = 2
Comm mode (0,1,2)        = 0,1,2
Dump flag                = 0
Elitist strategy         = 1
Traceflag                = 0
Global Select flag       = 0
Immigration rate         = 1
Naturalist flag          = 0
```

Figure 5.2 Parameter Settings for Parallel SGA

of messy GAs in general is predicated on the primordial phase ending with all the building blocks necessary to create an optimal solution. The cut probability is scaled linearly by current string length and is chosen such that strings at the fully specified length have a 50% chance of being cut and strings that are twice the fully specified length will always be cut.

Two important observations were made during the course of this investigation. First, while viewing some debugging output during the parallelization of the code, we noticed that each processor was sending multiple copies of only one or two building blocks into the global juxtapositional phase. This was inconsistent with the intent of the primordial phase. It was discovered to be the result of a poor building block filtering schedule which allowed aggressive competition between dissimilar building blocks. Although not an exhaustive test, three alternate schedules were proposed and tested against the original schedule. All three new schedules consisted of string-length reductions based on Goldberg's theory (36:60–61). The associated thresholding values chosen were:

1. 50% initial similarity, linearly increasing to 100% similarity at the last BBF episode
2. 50% initial similarity, linearly increasing to 80% similarity at the last BBF episode
3. constant 80% similarity enforced throughout the primordial phase

Table 5.5 Average Energy for Alternate Building Block Filtering Schedules

Schedule	Energy (kcal/mol)
Original	-14.5832
50% - 100%	-15.3590
50% - 80%	-17.5489
80%	-17.7322

Table 5.5 compares the average energy of all test runs for each new schedule and the original. (Individual test data is presented in Appendix A along with a sample output showing the building blocks from a typical run.) Based on this small test, the constant 80% schedule was chosen for the remainder of the investigation.

The second observation involves Goldberg's population sizing equation and the fmGA's use of tournament selection, as pointed out by Merkle (35, 63). The major difficulty with using population sizes calculated by Equation 5.1 for the protein folding problem is that the variance of all possible conformational energies is so large. However, we claim that since tournament selection is a direct competition selection operator, we can perform any *order-preserving* transformation on the fitness values without changing the outcome of a fmGA run. If this order-preserving transformation also reduces the variance of the fitness function, this would imply that the population size required to make correct selection decisions is smaller than estimated by the original application of the formula. Repeated applications of the order-preserving transformation should reach a "fixed point" that would be the true population size required by the fmGA. The transformation we choose to apply is $log_2(x + 1.0)$, where x is the value being transformed (fitness function maximum and signal difference value). Appendix B contains the short 'C' program used to carry out the transformations and the output that indicates convergence to a population size of 4,512 after 58 iterations. We opt for a slightly smaller maximum population size of 4,096 so we can be sure of the exact subpopulation size when multiple processors are used and keep execution times within reasonable limits. While this choice may limit the effectiveness of the algorithm, we are only concerned with evaluating the expected future worth of research in this area, not a final measurement of their best performance.

The parallel fmGA is exercised in the three different configurations described in Section 5.4.1. Ten iterations of each strategy, using different random seeds, are run on meshes that are powers of

two from 2 thru 7. Figure 5.3 represents a general input file for the parallel fmGA. Values separated by commas indicate the values that each parameter may assume in the experimental runs.

5.5 Summary

The test designs outlined in this chapter support the three major goals of this investigation: validate an enhanced CHARMm energy model, identify parameter sets that enable a simple GA to perform its best on our energy minimization problem, and compare the performance of simple and fast messy GAs, both serial and parallel. The results from these experiments should provide general insights into our energy model and the behavior of genetic algorithms attempting to optimize a real-world problem. The next chapter presents and analyzes the results of these experiments.

```
                  Random Seed = 1340988495
                  Experiments = 10
                String Length = 240
Block Size (1 - String_Length) = 5
                Genic Alphabet = 01
           Shuffle Number (>1) = 0
               Cut Probability = 0.002
            Splice Probability = 1.0
       Primordial_Generations = 95
            Total_Generations = 107
              Overflow (>1.0) = 1.6
                          n_a = 58,116,231,461,922,1844,3687
                      flags = I,C,G

Cut_gen   Str_len   Threshold
   0       235       118
   5       189        99
  10       151        81
  15       122        67
  20        98        56
  25        78        46
  30        63        38
  35        50        31
  40        41        26
  45        33        21
  50        26        17
  55        21        14
  60        17        12
  65        14        10
  70        11         8
  75         9         7
  80         7         5
  85         6         5
  90         5         4
```

Figure 5.3 Parameter Settings for Parallel fmGA

VI. Experimental Results and Analysis

This chapter contains the results obtained from the experiments defined previously in Chapter V. The data is analyzed to identify possible cause and effect relationships and the statistical significance of those relationships. In many cases, an average is calculated using quantities from multiple experiments in a test class. These averages are used to characterize the behavior of the algorithms because of their nondeterministic nature. The average execution time and average solution quality are better indications of general performance than individual data points. The Kruskal-Wallis test is used to compare the different test classes. This test accounts for the sample variances implicitly in the calculation of the statistic. The sections in this chapter parallel the last three sections of Chapter V for ease of reference.

6.1 Energy Model Validation

Tables 6.1 through 6.3 compare the energy values obtained from the three energy models for the accepted energy minimum conformation, a near-optimal conformation, and a random conformation respectively (Section 5.2). AFIT's old energy model was modified to correctly encode and decode the independent variables to obtain the energy values shown in the tables. While the energy values from the new energy model don't exactly match the accepted QUANTA values, there is now a definite correlation between the respective values. This correlation was noticeably absent between the old energy model and QUANTA.

Analyzing the differences shown in the tables, we notice that our enhanced energy model values are always lower than the accepted values. Although there is some error in the bond, bond

Table 6.1 Energy Component Comparison, [Met]-enkephalin Native Conformation

Energy Component	Previous Implementation (kcal/mol)	QUANTA Values (kcal/mol)	New Implementation (kcal/mol)
Bond (ρ)	12.380	12.374	12.380
Bond Angle (θ)	6.187	6.183	6.189
Dihedral Angle (ϕ)	202.480	8.204	8.203
Lennard-Jones	-45.118	-15.014	-20.201
Electrostatic	-0.267	-40.973	-40.091
Total	175.662	-29.225	-33.520

Table 6.2 Energy Component Comparison, [Met]-enkephalin Near-Optimal Conformation

Energy Component	Previous Implementation (kcal/mol)	QUANTA Values (kcal/mol)	New Implementation (kcal/mol)
Bond (ρ)	12.381	12.311	12.380
Bond Angle (θ)	2491.940	6.210	6.189
Dihedral Angle (ϕ)	200.453	6.731	6.731
Lennard-Jones	2.266e+09	-13.112	-17.066
Electrostatic	-0.236	-38.066	-38.086
Total	2.266e+09	-25.927	-29.851

Table 6.3 Energy Component Comparison, [Met]-enkephalin Random Conformation

Energy Component	Previous Implementation (kcal/mol)	QUANTA Values (kcal/mol)	New Implementation (kcal/mol)
Bond (ρ)	12.378	12.378	12.380
Bond Angle (θ)	1535.251	6.198	6.189
Dihedral Angle (ϕ)	199.082	42.816	42.817
Lennard-Jones	2980447.172	70.090	38.090
Electrostatic	-0.252	-29.409	-29.361
Total	2982193.631	102.075	70.116

angle, and dihedral angle energy terms, the differences are at least two orders of magnitude smaller than the overall error, and thus are currently a minor concern. However, the error can be accounted for by observing that the binary encoding scheme used by our algorithms is a source of round-off error (Section 2.2).

The major source of error is in the non-bonded interaction energy term and more specifically, in the Lennard-Jones potential (Equation 3.1). After numerous communcations with the QUANTA technical support staff, we've determined a possible source of the difference. CHARMm has recently been changed to use E_{min} and R_{min} directly from the parameter file to calculate the Lennard-Jones potential. In contrast, we use these values to calculate the intermediate parameters A and B used in Equation 3.1 as described by Brooks *et. al.* (4). These parameters can be a major source of error since they are calculated by finding the 6th and 12th roots of numbers that are less than one!

Figure 6.1 represents the best conformation found by any of the four genetic algorithms. The energy terms calculated by AFIT's energy model and QUANTA are compared in Table 6.4. Figures 6.2 and 6.3 show the best GA conformation superimposed on the accepted minimum energy

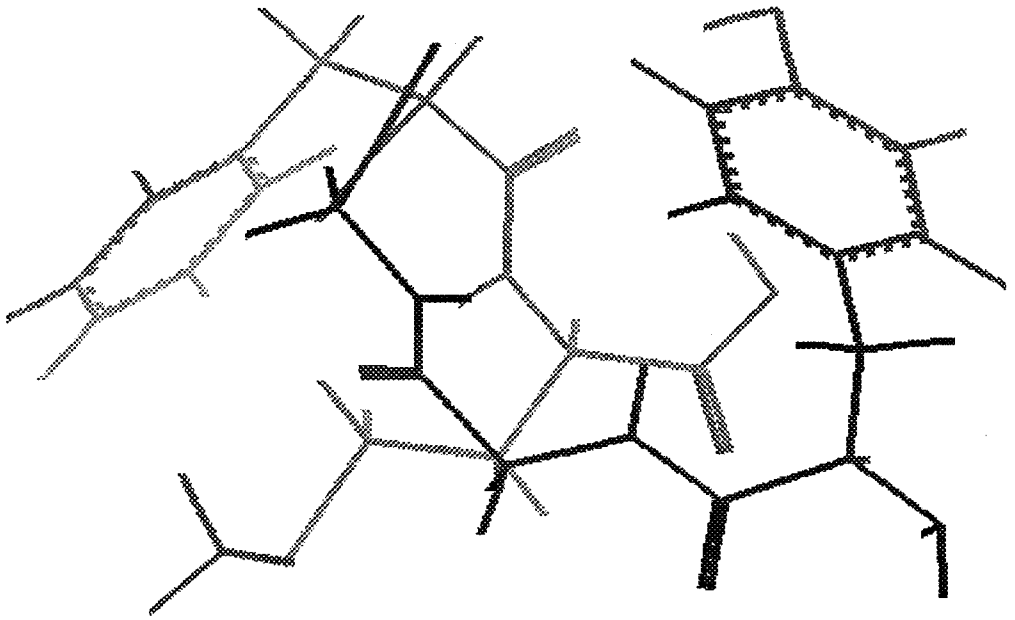


Figure 6.1 GA Minimized Conformation of [Met]-enkephalin

conformation of [Met]-enkephalin. The first view shows a very good match between the molecules. The second view is rotated approximately 90° from the first. Although the various dotted lines connecting similar atoms in the two conformations appear to indicate a poor match, their relatively parallel orientation, along with the first view, indicates a possible mirror image conformation. Table 6.5 compares the 24 independent dihedral angles of the GA solution with those of the accepted conformation. Interestingly, the shorter, internal amino acid dihedral angles are closer to their correct values than the longer residue dihedral angles on the ends of the molecule. This could be a result of the error in the non-bonded energy terms which would tend to exert its influence on the extremities of a molecule.

6.2 Simple Genetic Algorithm Parameters for Protein Energy Minimization

Table 6.6 lists the 23 members of the best online pool beginning with the parameter set that exhibits the best average online performance. The members of the pool were identified using the Kruskal-Wallis test (86:544–546). The statistic, calculated from the raw sample data, is $h = 31.691$. This value is less than the critical value of the Chi-square distribution at the $\alpha < .05$ significance

Table 6.4 Energy Component Comparison, [Met]-enkephalin GA Best Found Conformation

Energy Component	QUANTA Values (kcal/mol)	New Implementation (kcal/mol)
Bond (ρ)	12.443	12.380
Bond Angle (θ)	6.200	6.189
Dihedral Angle (ϕ)	9.359	7.786
Lennard-Jones	-15.592	-20.383
Electrostatic	-40.602	-42.335
Total	-28.191	-36.362

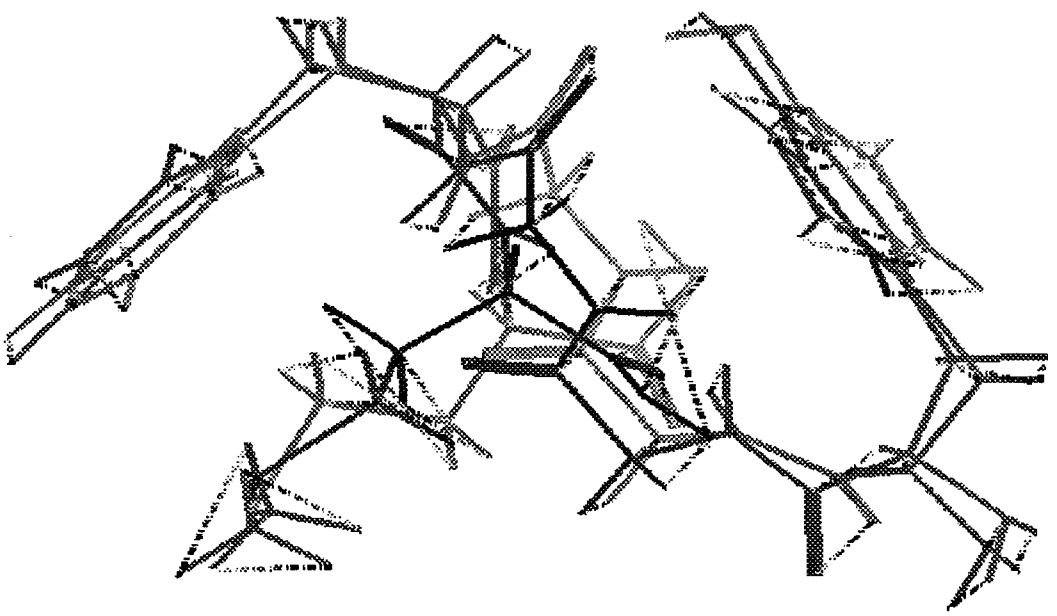


Figure 6.2 Superimposed Conformations of [Met]-enkephalin (View #1)

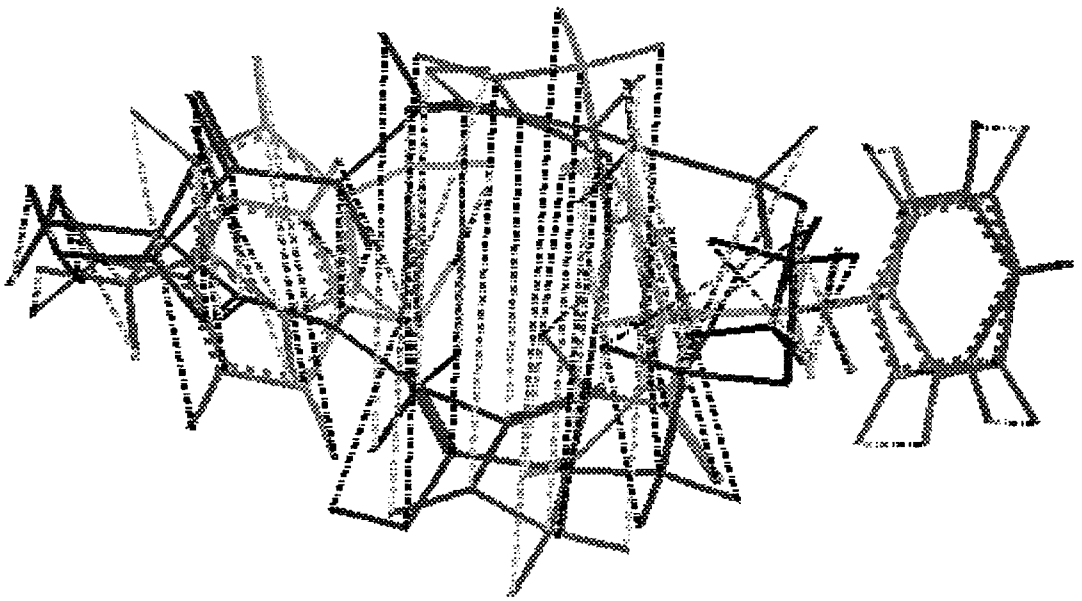


Figure 6.3 Superimposed Conformations of [Met]-enkephalin (View #2)

Table 6.5 Dihedral Angle Comparison, [Met]-enkephalin GA Best Found Conformation

	Dihedral Angle (degrees)						
Residue	ϕ	ψ	ω	χ^1	χ^2	χ^3	χ^4
Tyr							
GA	-62.9	140.3	-179.3	-180.0	69.3	-62.6	
Accepted	-86.1	156.1	-176.8	-172.6	78.8	165.9	
Gly							
GA	-150.8	80.2	170.9				
Accepted	-154.3	83.7	168.8				
Gly							
GA	78.8	-84.4	177.2				
Accepted	83.7	-73.8	-170.2				
Phe							
GA	-103.7	-10.9	-179.3	44.3	-93.9		
Accepted	-137.1	19.3	-174.0	58.7	-85.4		
Met							
GA	-78,4	69.6	-9.5	-57.3	-173.7	90.0	-173.3
Accepted	-163.5	160.3	-179.7	52.7	175.1	-180.0	-58.4

Table 6.6 Best Online Pool

Population Size	Crossover Rate	Mutation Rate	Mean Online Performance
10	0.75	0.005	0.014678
10	0.85	0.005	0.014678
10	0.95	0.005	0.014920
20	0.95	0.002	0.015697
20	0.65	0.002	0.015831
20	0.75	0.002	0.017045
10	0.55	0.005	0.017276
10	0.65	0.005	0.017276
20	0.85	0.002	0.017567
20	0.35	0.002	0.017919
30	0.15	0.002	0.018720
20	0.25	0.005	0.019704
20	0.45	0.002	0.020177
10	0.15	0.005	0.021270
10	0.25	0.005	0.021270
20	0.55	0.002	0.021707
30	0.05	0.002	0.022144
20	0.55	0.005	0.022557
10	0.05	0.005	0.022617
20	0.15	0.002	0.022739
10	0.35	0.005	0.023407
10	0.45	0.005	0.023407
20	0.25	0.002	0.023686

level with 22 degrees of freedom. Thus we accept the null hypothesis and conclude that these 23 parameter settings exhibit the same online performance!

Figures 6.4 through 6.9 show the average online performance for all 420 parameter set combinations. The diamonds in the graphs mark the locations of the members of the best online pool. The results show relationships between population size, mutation rates, and crossover rates similar to those reported by Schaffer (77:54–59). Included in these relationships is a very strong correlation between population size and mutation rate. Alternately, there is no evidence of any relationship between population size and crossover probability.

As discussed by Dymek (20), premature convergence is a major problem for genetic algorithms. The graphs in Figures 6.4 through 6.6 show pictorially that premature convergence is

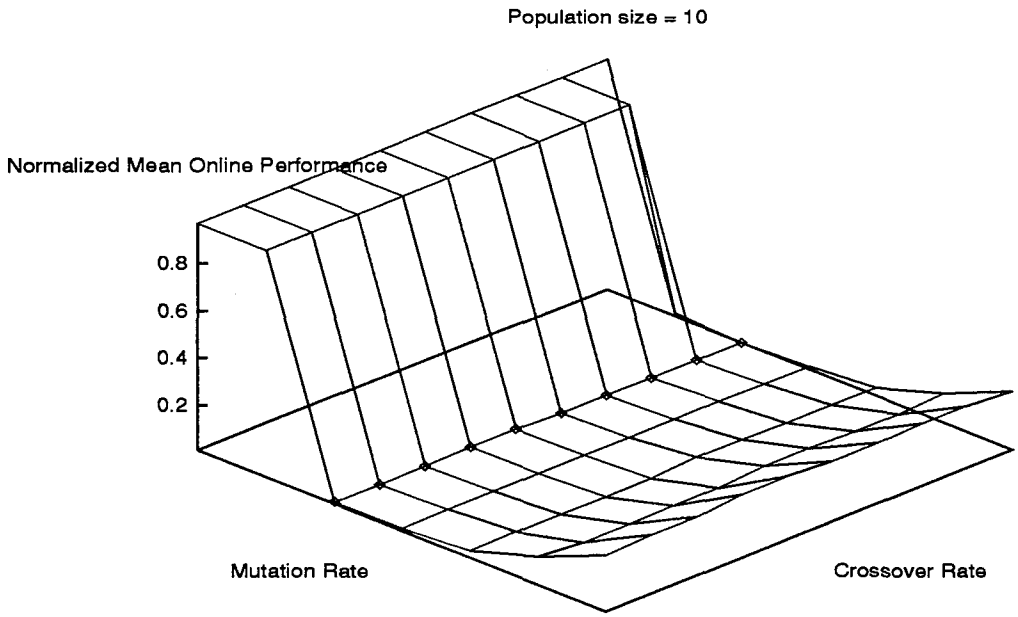


Figure 6.4 Mean Online Performance (Population Size = 10)

highly likely when the mutation rate is set too low for the chosen population size. Every point in the graphs that represents a normalized mean online performance near 1.0 also represents a GA execution that prematurely converged. On the other side of the best online pool, as the mutation rate increases, the GAs perform successively more random searches. Searches using these parameters are less likely to exhibit good online performance because they aren't focused on exploiting the good solutions already found.

6.3 Evaluation of SGAs and fmGAs for the Protein Folding Problem

6.3.1 Parallel Simple GAs. Figure 6.10 shows average solution quality improving as the number of processors is increased and the global population size remains fixed at 640 using the parallel SGA. This result is expected based on the previous findings that small populations (in this case subpopulations) exhibit better online performance than large populations! Analysis of the raw data indicates the results from the three communcation strategies are statistically the same at the $\alpha < .05$ significance level (32).

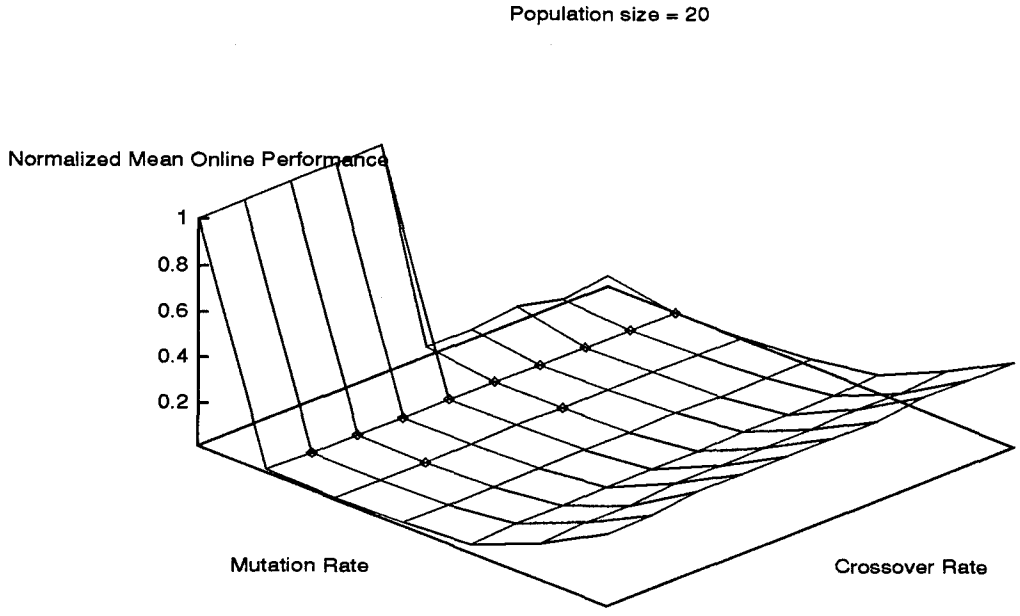


Figure 6.5 Mean Online Performance (Population Size = 20)

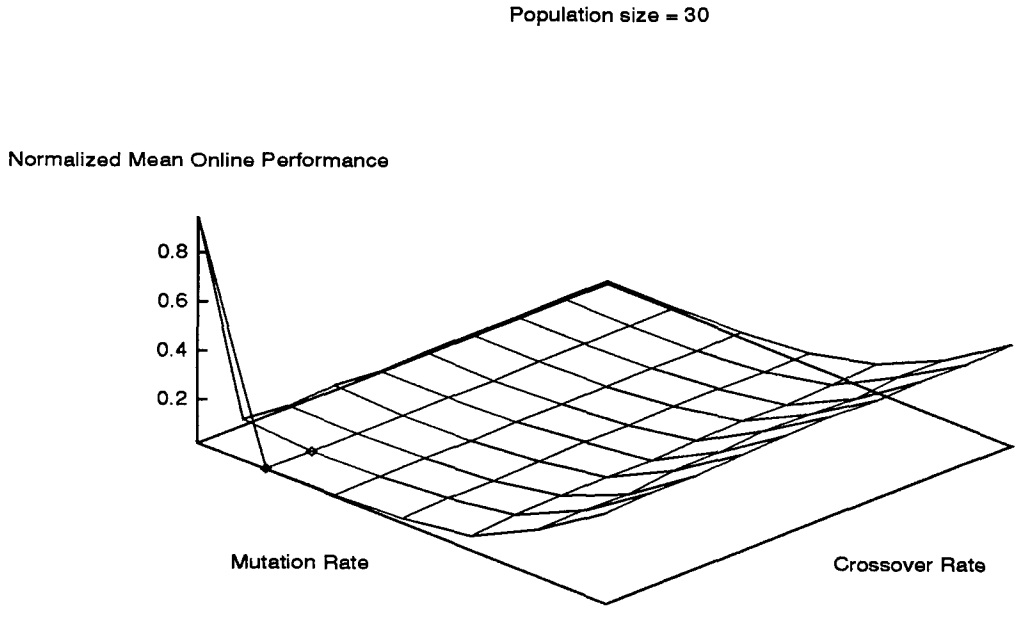


Figure 6.6 Mean Online Performance (Population Size = 30)

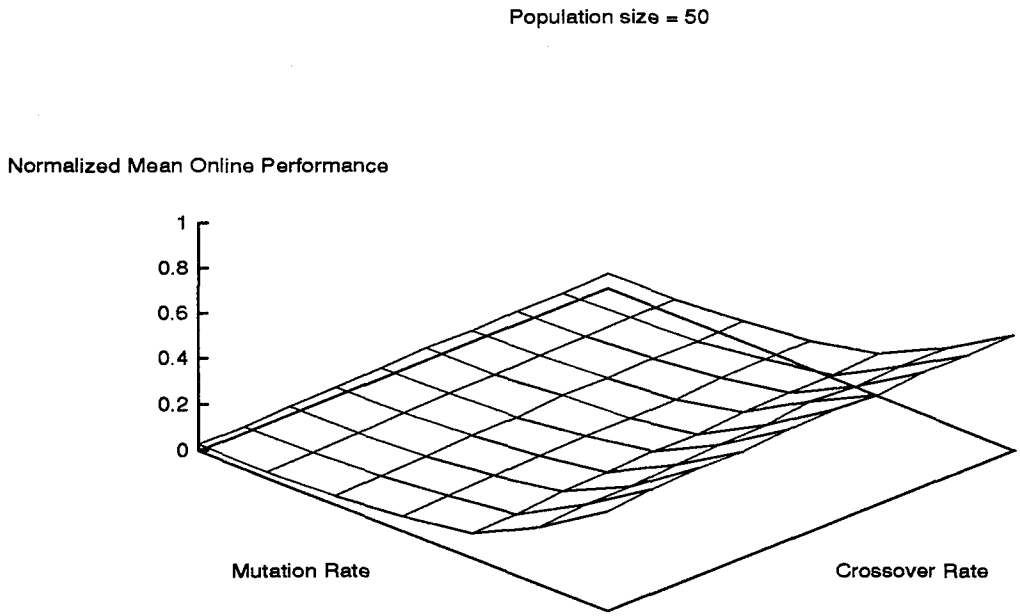


Figure 6.7 Mean Online Performance (Population Size = 50)

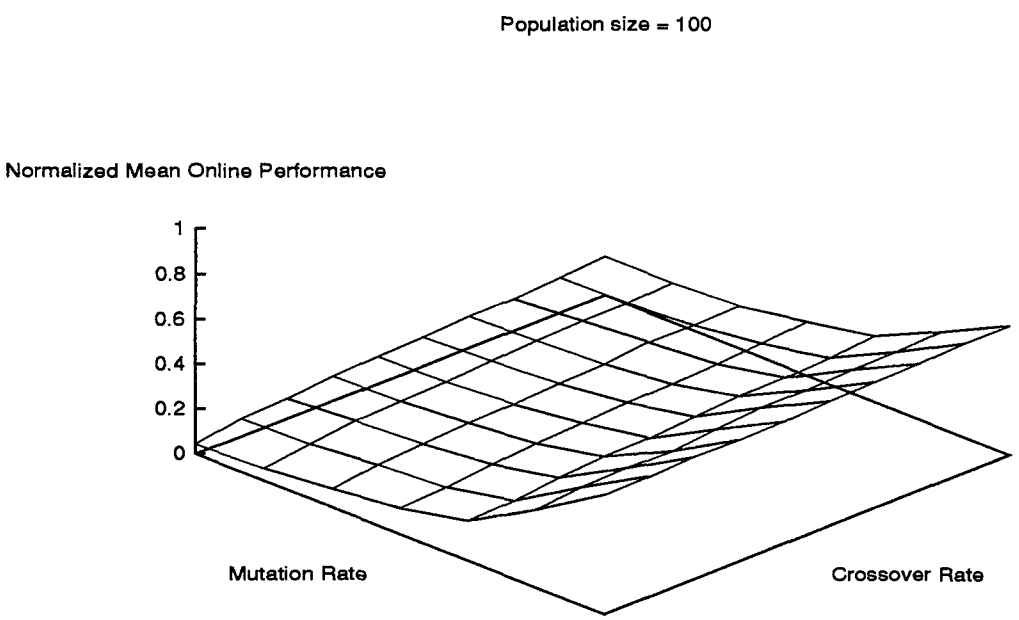


Figure 6.8 Mean Online Performance (Population Size = 100)

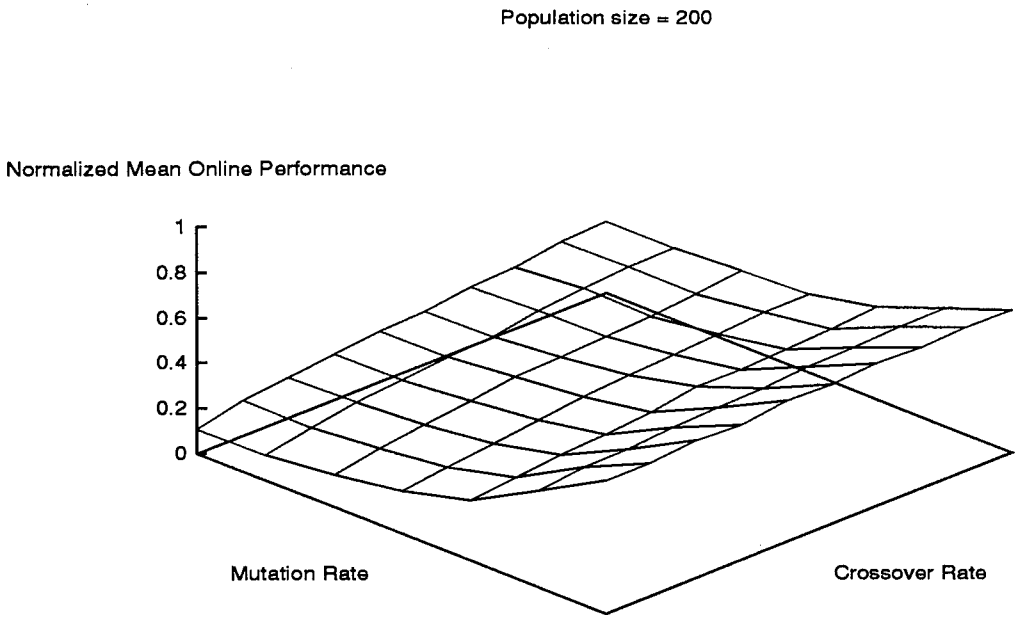


Figure 6.9 Mean Online Performance (Population Size = 200)

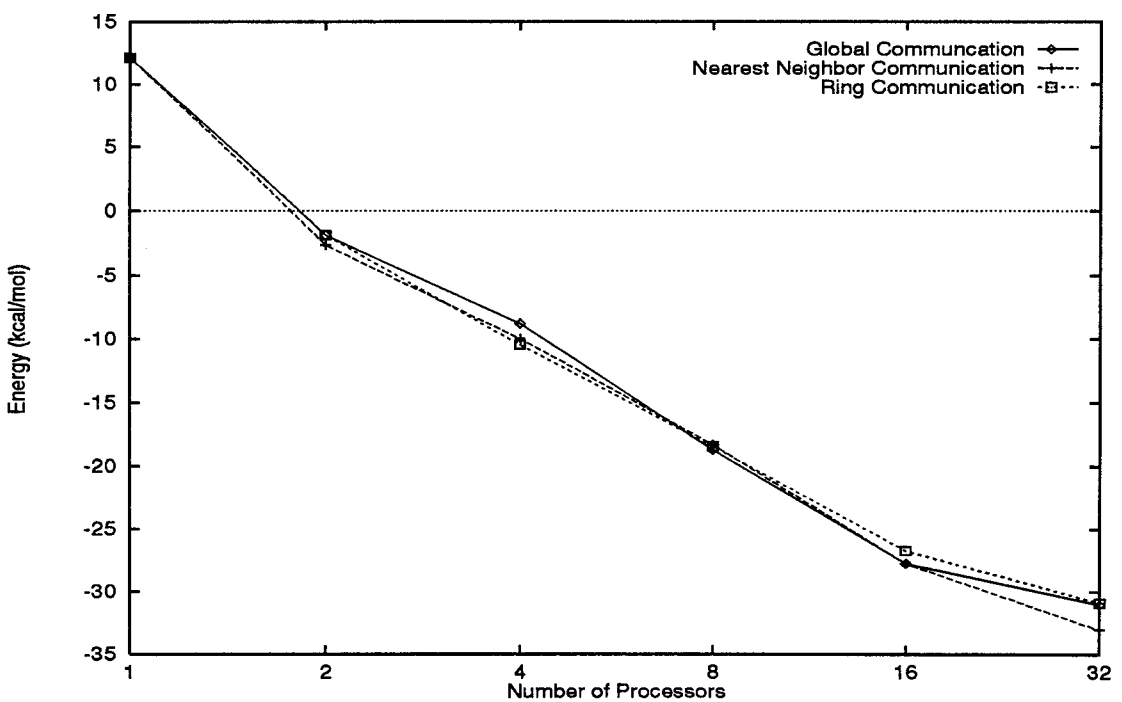


Figure 6.10 Parallel SGA Average Minimum Energy (Global Population Size Fixed at 640)

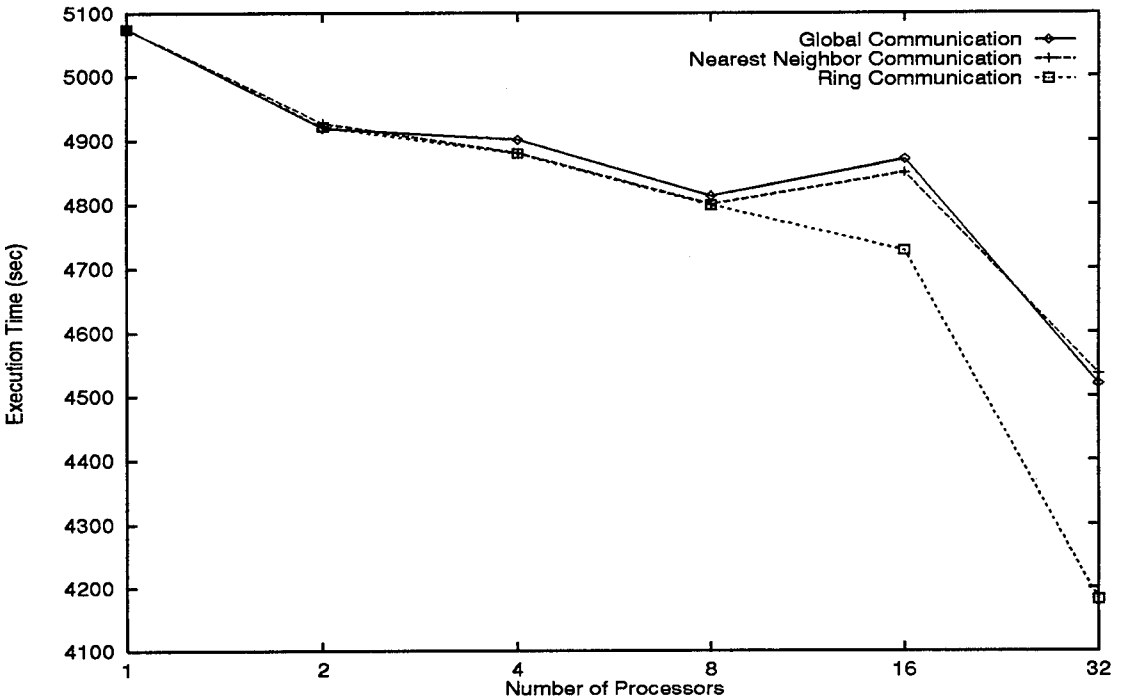


Figure 6.11 Parallel SGA Average Execution Time (Global Population Size Fixed at 640)

Figure 6.11 plots the average execution times associated with the solutions from Figure 6.10. Given that there is no statistical difference in solution quality between the three communication strategies, the strategy of choice is ring communication because of its significantly lower execution times!

Figure 6.12 shows the calculated speedup based on the observed average execution times. If the same amount of work is being performed, the observed superlinear speedup should be impossible! An analysis of our parallel SGA can explain this phenomenon although the output that could validate this hypothesis cannot be obtained from the current implementation. First, the number of generations is the primary halting condition for the parallel SGA. Second, the fitness of each string is retained so that it doesn't need to be recomputed if a string remains unchanged from one generation to the next. Combining these observations with the results from Section 6.2 and Merkle's MS thesis (62:110–114), the smaller subpopulations used with more processors tend to converge faster. Faster convergence results in fewer fitness evaluation calculations, and thus less work is performed.

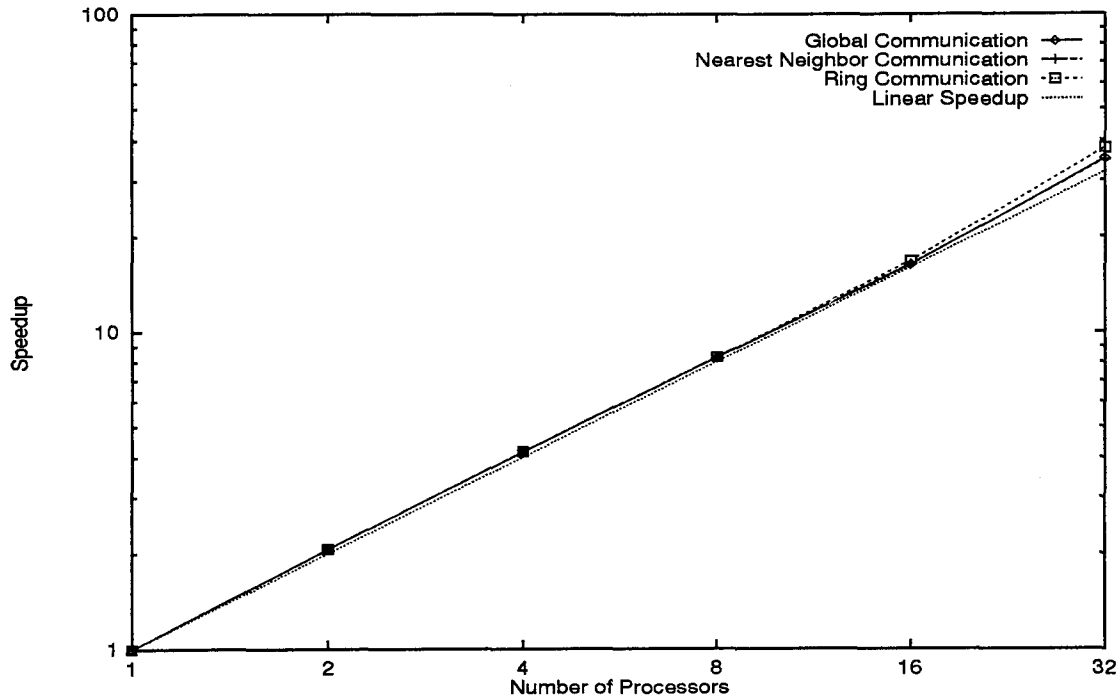


Figure 6.12 Parallel SGA Speedup (Global Population Size Fixed at 640)

Figure 6.13 shows average solution quality improving as the number of processors is increased and the subpopulation size remains fixed at 20 using the parallel SGA. The results support the conclusion that parallel SGAs using these solution sharing strategies are capable of finding significantly better solutions than independently executed SGAs. Again, the results from the three communcation strategies are statistically indistinguishable at the $\alpha < .05$ significance level (32).

Figure 6.14 plots the average execution times associated with the solutions from Figure 6.13. Since there is no statistical difference in solution quality between the three communication strategies, the ring communication is again the strategy of choice because of its significantly lower execution times.

Strictly speaking, we can't calculate speedup for this execution time data because we are effectively doubling the amount of work each time we double the number of processors. However, we can calculate an upper bound on the *scaled speedup* (54:144). To do this we simply divide the parallel execution time by the relative size of the workload before using the normal speedup calculation. Figure 6.15 shows the calculated scaled speedup based on the observed average execution times. All

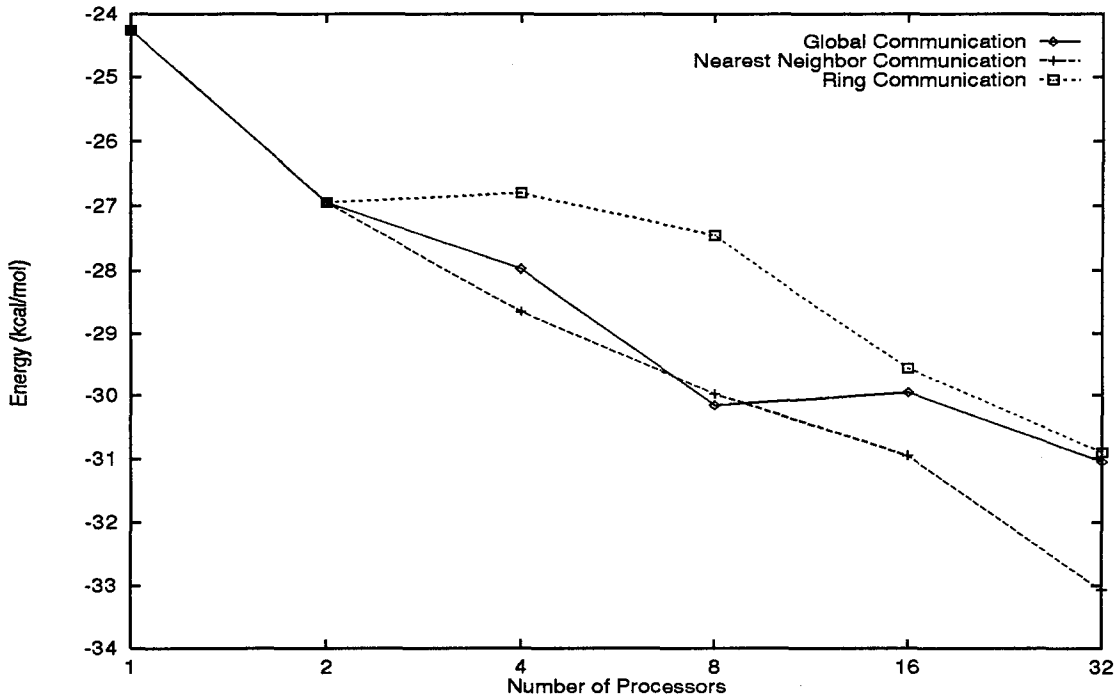


Figure 6.13 Parallel SGA Average Minimum Energy (Subpopulation Size Fixed at 20)

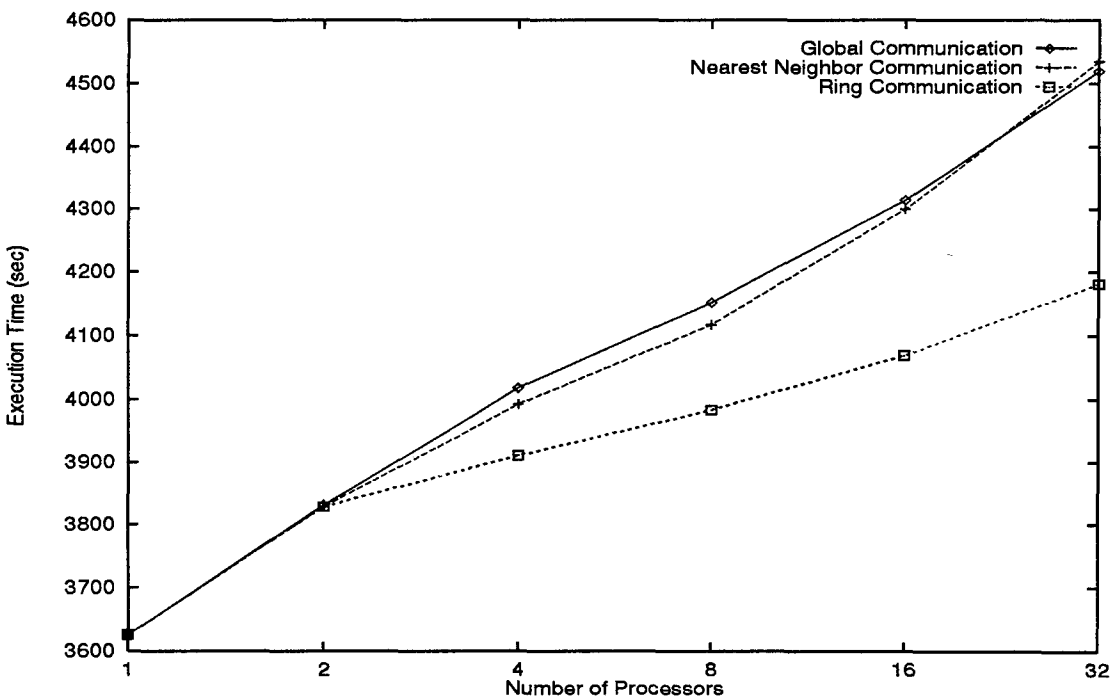


Figure 6.14 Parallel SGA Average Execution Time (Subpopulation Size Fixed at 20)

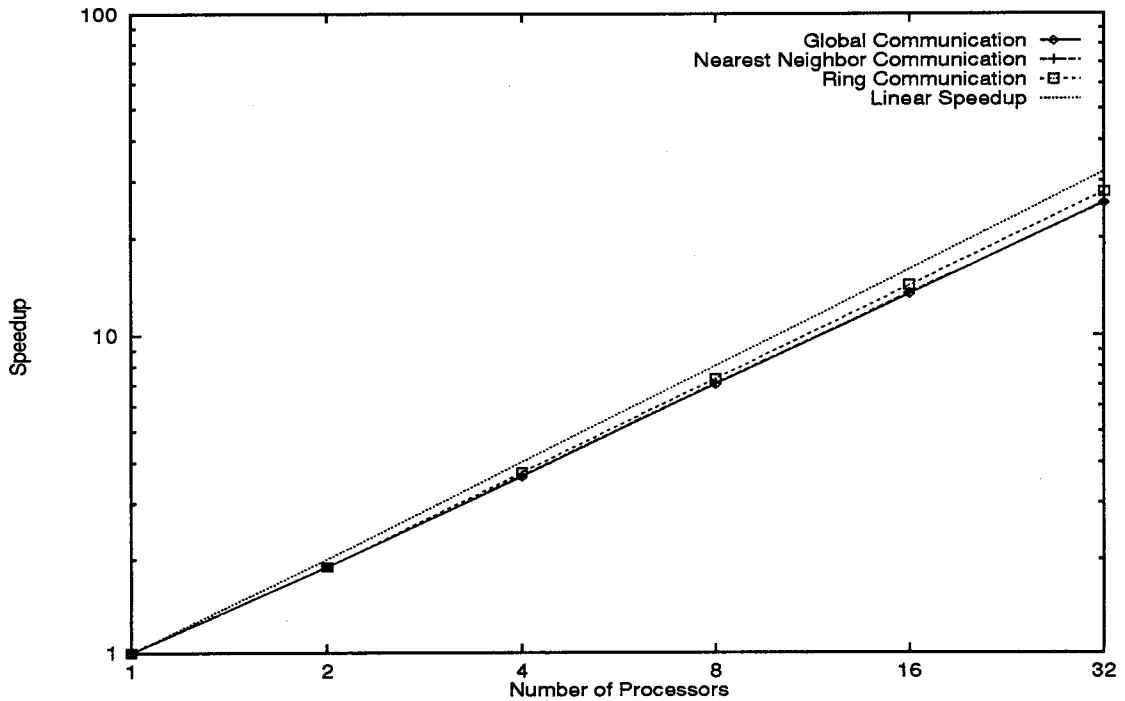


Figure 6.15 Parallel SGA Speedup (Subpopulation Size Fixed at 20)

three communication strategies exhibit near-linear speedup with the ring communication strategy performing slightly better than global and nearest neighbor communication strategies.

From Figures 6.10 and 6.13 we see that the fixed subpopulation parallel SGA always finds better solutions than the fixed global population parallel SGA. Figures 6.11 and 6.14 also show that the fixed subpopulation parallel SGA always finds the solution faster too! Based on the scaled speedup shown in Figure 6.15, the efficiency of the fixed subpopulation parallel SGA is plotted in Figure 6.16.

6.3.2 Parallel fmGAs. Figure 6.17 shows average solution quality generally improving as the number of processors is increased and the subpopulation size remains fixed at 32 using the parallel fmGA (32). Since no other parameters are changing, these results support the conclusion that parallel fmGAs obtain better results with larger population sizes. This conclusion supports Goldberg's population sizing calculations for fmGAs (35) and the order-preserving transformation of that calculation discussed in Section 5.4.2.

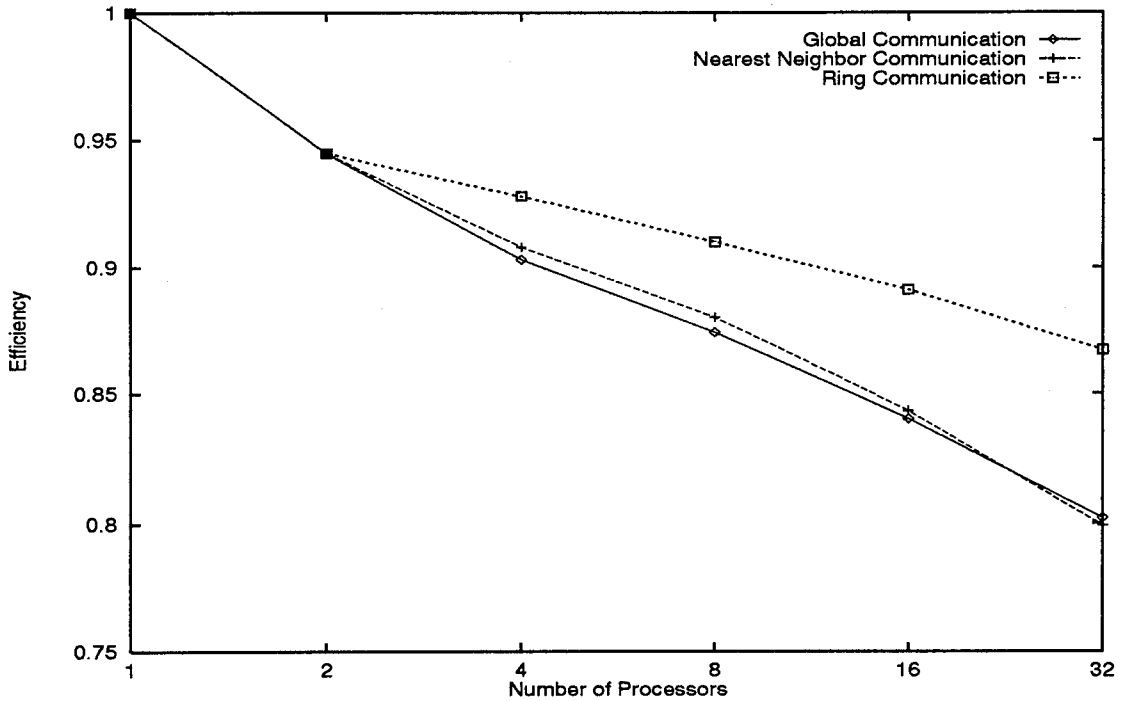


Figure 6.16 Parallel SGA Efficiency (Subpopulation Size Fixed at 20)

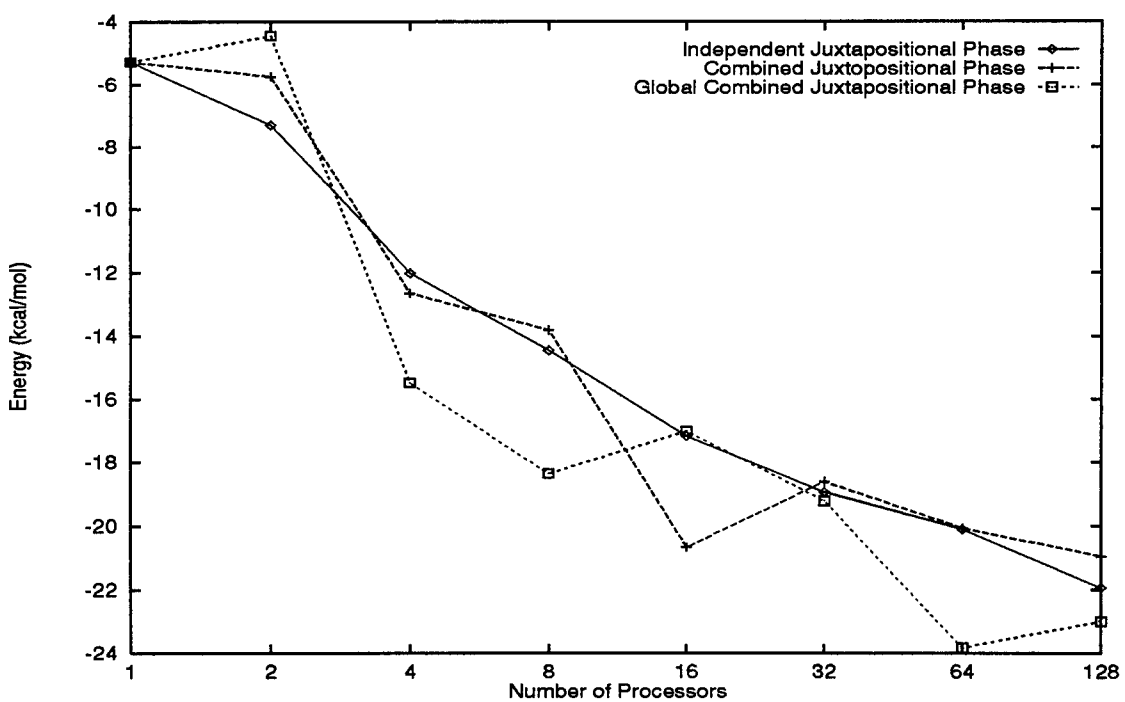


Figure 6.17 Parallel fmGA Average Minimum Energy (Subpopulation Size Fixed at 32)

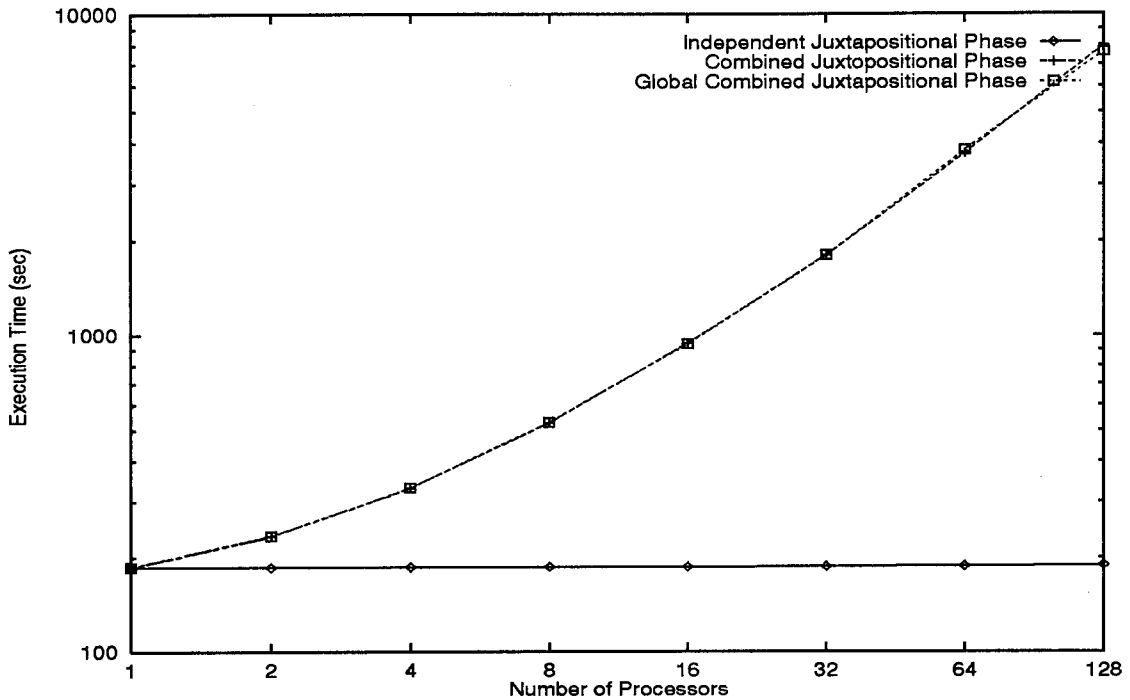


Figure 6.18 Parallel fmGA Average Execution Time (Subpopulation Size Fixed at 32)

Figure 6.18 plots the average execution times associated with the solutions from Figure 6.17. No communcation strategy can be chosen as the best from this experiment because the independent juxtapositional phase strategy exhibits the best execution time performance, but it cannot generally finds superior solutions. The other two strategies show a very limited ability to find better solutions, however the significantly longer execution times prohibit their practical use.

Using a fixed subpopulation size we have to use the scaled speedup calculation again because our workload is linearly proportional to the number of processors. Figure 6.19 shows the calculated scaled speedup based on the observed average execution times. Only the independent juxtapositional phase strategy exhibits near-linear speedup.

Figure 6.20 shows average solution quality which is statistically indestinguishable up through eight processors. From 16 processors up through 128, the global combined juxtapositional phase consistently outperforms the other two communication strategies (32). The energy values are consistantly better than those shown in Figure 6.17. These results support the conclusion that parallel fmGAs obtain better results with larger population sizes.

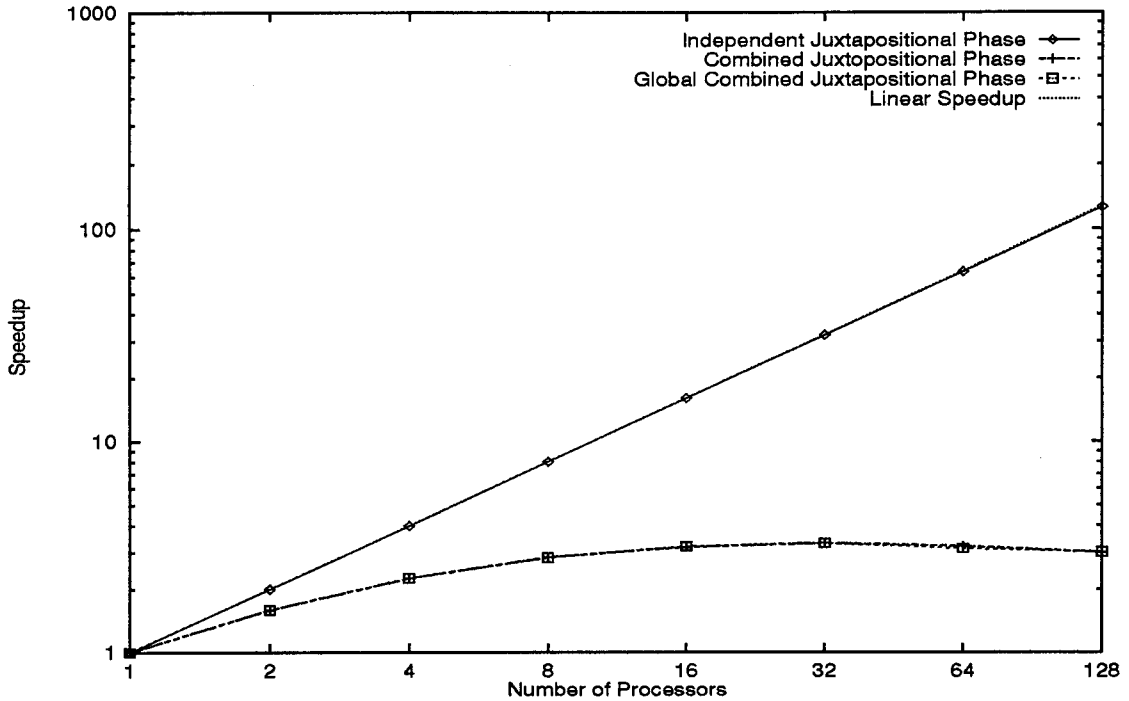


Figure 6.19 Parallel fmGA Speedup (Subpopulation Size Fixed at 32)

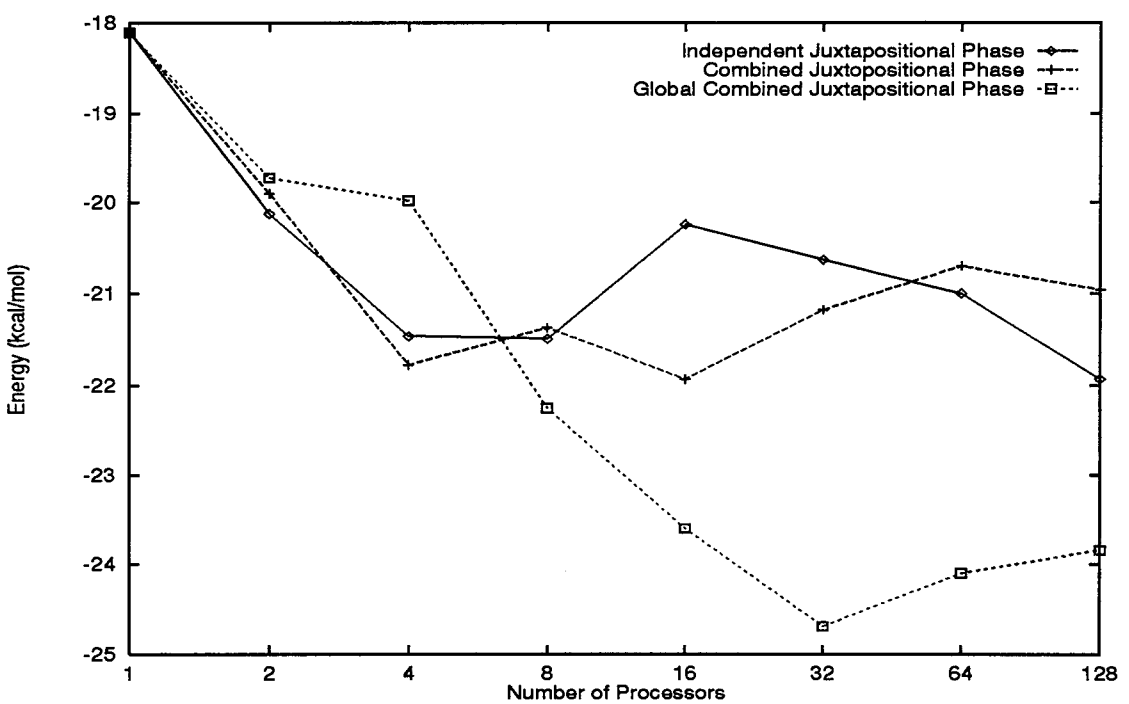


Figure 6.20 Parallel fmGA Average Minimum Energy (Global Population Size Fixed at 4096)

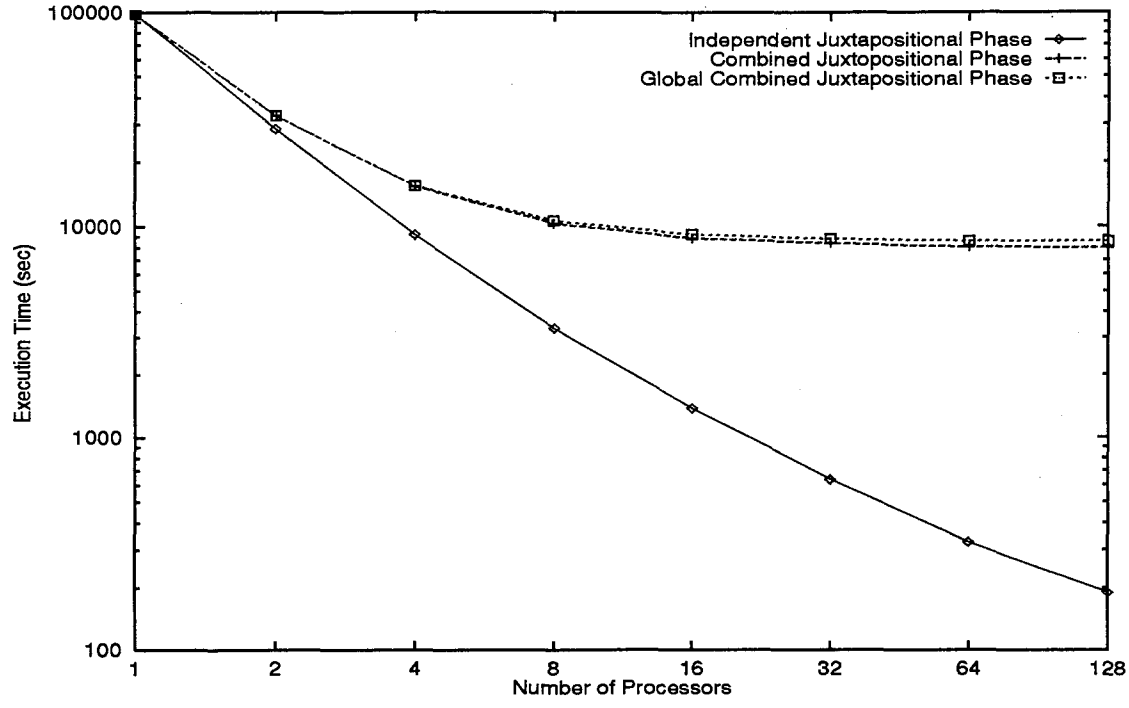


Figure 6.21 Parallel fmGA Average Execution Time (Global Population Size Fixed at 4096)

Figure 6.21 plots the average execution times associated with the solutions from Figure 6.20. Although the global combined juxtapositional phase strategy finds much better solutions than the other two strategies for large numbers of processors, the execution time is an order of magnitude larger than the independent juxtapositional phase strategy.

Figure 6.22 shows the calculated speedup based on the observed average execution times. Again, if the same amount of work is being performed, the observed superlinear speedup should be impossible. An analysis of our parallel fmGA can explain this behavior. There is a parameter called *shuffle number* in the fmGA that determines how many members of the population are compared to find compatible strings for competition during tournament selection. By default, this parameter is set to force a search through the entire population (n) which would make the execution time for one generation $\mathcal{O}(n^2)$. Thus, when the subpopulation size is halved, the execution time of this part of the algorithm is quartered and the exhibited superlinear speedup is observed.

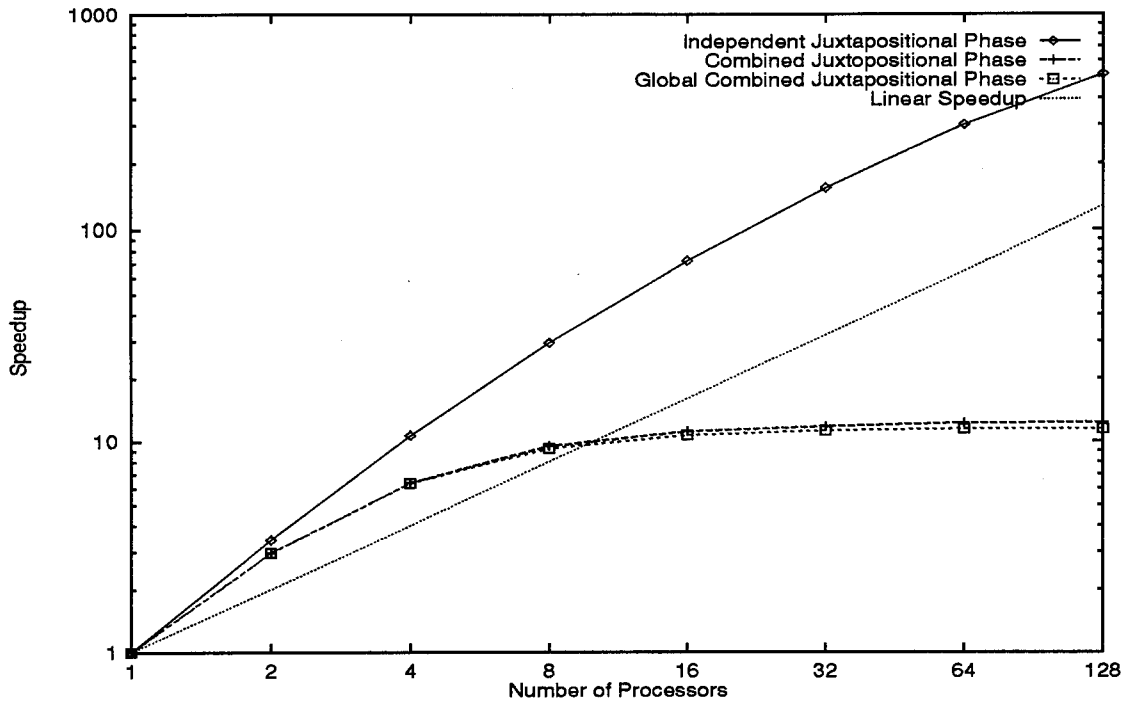


Figure 6.22 Parallel fmGA Speedup (Global Population Size Fixed at 4096)

6.4 Summary

Chapter I proposes three main objectives for this investigation. This chapter presents the empirical results from the experiments designed in Chapter V to meet those objectives. The accuracy of the enhanced energy model is analyzed and possible sources of error identified. The effects of parameter set choices are observed and found to corroborate previous empirical results. Finally, the performance of simple and fast messy GAs is compared using several efficiency and effectiveness metrics.

VII. Conclusions and Future Directions

Based on the literature review, design, implementation, and experimental work discussed previously as well as general observations throughout this research effort, the major conclusions and recommendations for future research are presented. The conclusions are drawn from the work accomplished to meet the three primary objectives of this investigation: validate AFIT's energy model, identify good SGA parameters, and compare SGA and fmGA performance.

7.1 Conclusions

Our energy model is now more reliable as a tool to minimize the potential energy of polypeptides. The bonded, bond angle, and dihedral angle energy terms are now in agreement with the values calculated by QUANTA. Although the non-bonded energy term values aren't the same as QUANTA's, the relative order of energy values is maintained by the new model. Near the accepted minimum, the energy model tends to underestimate the non-bonded energy contribution by approximately four kcal/mol. This is a vast improvement over AFIT's original energy model (3).

The parameter settings that enable the SGA to perform the best optimization of [Met]-enkephalin's energy potential fall within the ranges observed by Schaffer for general function optimization (77). From these results we can be fairly confident that we've observed the SGA's "best" search performance on this optimization problem. The results also confirm that choosing parameters (population size, mutation rate, crossover rate) within the ranges specified by Schaffer (10–30, 0.005–0.1, 0.65–0.95) is a good starting point. SGA online performance has been shown to be extremely sensitive to the combined choice of population size and mutation rate (see Section 6.2).

Currently, the genetic algorithm of choice for minimizing the energy of polypeptides should be the parallel SGA using a small, constant subpopulation size and the ring communication strategy. Using parameters from the best online pool, this GA exhibits good scaled speedup (Figure 6.15) and efficiency (Figure 6.16) and finds better solutions than any of the other algorithm/communication strategy/parameter combinations (Figure 6.13). However, the fmGA cannot be eliminated from consideration by this single, preliminary application. The parallel fmGA with a constant global population size actually exhibited decent performance considering:

- It is currently unknown what constitutes a good building block filtering schedule
- It is unclear how to estimate the deception associated with specific problem domains
- Our competitive template isn't optimal at the $k-1$ block size
- The communications algorithms are inefficiently implemented because of the current data structures
- Many of the low-level routines are inefficiently implemented

7.2 Future Research Recommendations

Many possibilities exist for future research in protein structure prediction using genetic algorithms. This section presents some of those possibilities that would logically follow from this investigation. The recommendations fall into three general categories: continued refinement of AFIT's energy model, leading edge research of fast messy GAs, and identification and evaluation of alternative optimization approaches.

The non-linear error in the non-bonded energy term of AFIT's energy model needs to be isolated and corrected. The remaining differences cannot be resolved from the QUANTA documentation or the CHARMm paper by Brooks *et. al.* (8, 4). This will require close contact with the people at Harvard University responsible for the CHARMm energy model. As the differences between the two models becomes smaller, it is increasingly likely that errors may be found in either implementation. Thus the collaboration may result in a better energy model for all interested parties.

Based on the strong correlation between the two energy model implementations, AFIT's energy model is now "robust enough" to attempt minimizing the energy of larger polypeptides. This work should provide additional insight into the performance and scalability of genetic algorithms applied to the protein folding problem in general.

There are many research areas that must be addressed to bring the level of maturity of fast messy GAs up to par with SGAs. An investigation of building block filtering (BBF) schedules should be the top priority. The identification of *good* BBF schedules is necessary to ensure that the primordial phase performs its intended function. The efficient operation of the juxtapositional phase is dependent on the quality and quantity of highly fit building blocks it receives from the primordial phase (see Section 2.4).

Although it has not been used in any work to date, the original fmGA design also specifies an *outer loop* that can be used to step through building block sizes from 1 to a specified maximum block size. During iteration $x+1$ the order-x optimal solution from the previous iteration is used as the competitive template. The performance associated with this outer loop needs to be evaluated.

Modifying building block filtering to be a stochastic operation might be another viable alternative to the previously described iterative method. In the spirit of GA search, the stochastic method is expected to simultaneously search for the correct linkage and alleles like the current fmGA as well as the building block sizes. It is speculated that this approach may be more efficient than the iterative method and it could possibly eliminate the need to choose a maximum expected level of deception for every problem instance.

Several hybridization techniques may also increase the efficiency and effectiveness of genetic algorithms in general. Examples of these combined approaches may include:

- Using local minimization as an additional genetic operator
- Integrating deterministic and/or stochastic optimization techniques as cooperative agents
- Pipelining several techniques based on their abilities to find/refine solutions

Performance comparisons need to be made between the current binary encoding scheme approaches and higher cardinality implementations. That investigation should include real-valued GAs and other evolutionary algorithms that have their basis in real-valued parameter optimization (evolutionary strategies and evolutionary programming).

In addition to the identified topics for future research, the following work is suggested as ideas for course projects. The parallel SGA code needs to be redesigned and implemented for clarity to enhance our configuration control capability. The current implementation is tightly coupled with the problem domain and exhibits very low cohesion and little modularity (76). The parallel SGA code should also be ported to the Intel Paragon and other massively parallel computer architectures to enable larger scalability investigations and because our access to Intel hypercubes is quickly diminishing.

There is a lot of optimization work that needs to be accomplished if these algorithms are to be used on larger proteins. Several data structures need to be changed to efficiently use the global system routines on the Paragon. More efficient communication can also be realized with

data structure improvements. In general the GA string data structures need to be allocated in contiguous memory so they can be quickly communicated as a unit. Other possible enhancements include using the system math libraries for matrix and vector operations and in-lining the functions that are evaluated repeatedly (evaluation function, overlay function, thresholding function).

7.3 Summary

This chapter summarizes the general conclusions that can be drawn from this investigation. These conclusions are used to highlight critical areas of future research related to protein structure prediction and genetic algorithms. Overall, this thesis documents the successes and failures associated with: enhancing the accuracy of AFIT's energy minimization model; identifying parameter settings that encourage good SGA online performance; and evaluating the performance and maturity of serial/parallel, simple/fast messy GAs.

Appendix A. Building Block Filtering Schedule Test Data

Section 5.4.2 describes a small experiment designed to find a better building block filtering schedule. This appendix reports the raw data gathered from those experiments.

A.1 Energy Values

Table A.1 shows the energy values obtained from five runs of each building block filtering schedule.

Table A.1 Raw Energy Values from Alternate Building Block Filtering Schedules

Schedule	Trial				
	#1	#2	#3	#4	#5
Original	-14.5832	-14.5832	-14.5832	-14.5832	-14.5832
50% - 100%	-15.4414	-15.9923	-15.4414	-15.4414	-14.4784
50% - 80%	-16.3489	-16.4705	-16.3489	-18.8422	-19.7338
80%	-17.9176	-17.9176	-17.9176	-17.9176	-16.9906

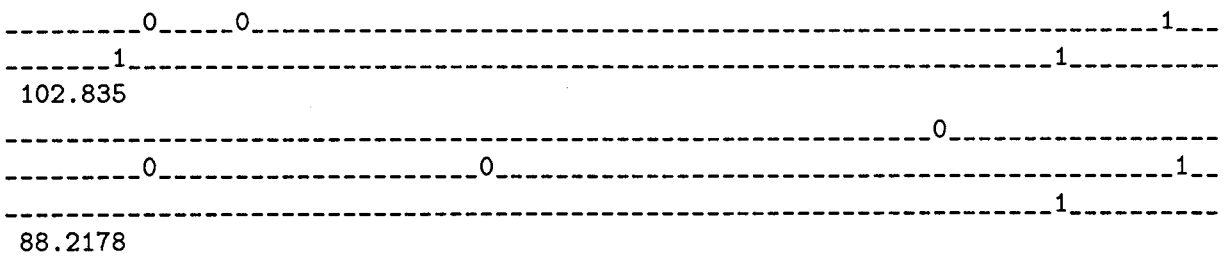
A.2 Sample Building Blocks

The two following sections show the building blocks that are created during the primordial phase using the original building block filtering schedule and the selected constant 80% schedule respectively. The lack of a diverse population of building blocks explains the poor performance of the juxtapositional phase using the original schedule. This schedule only generates three unique building blocks out of a population of 64.

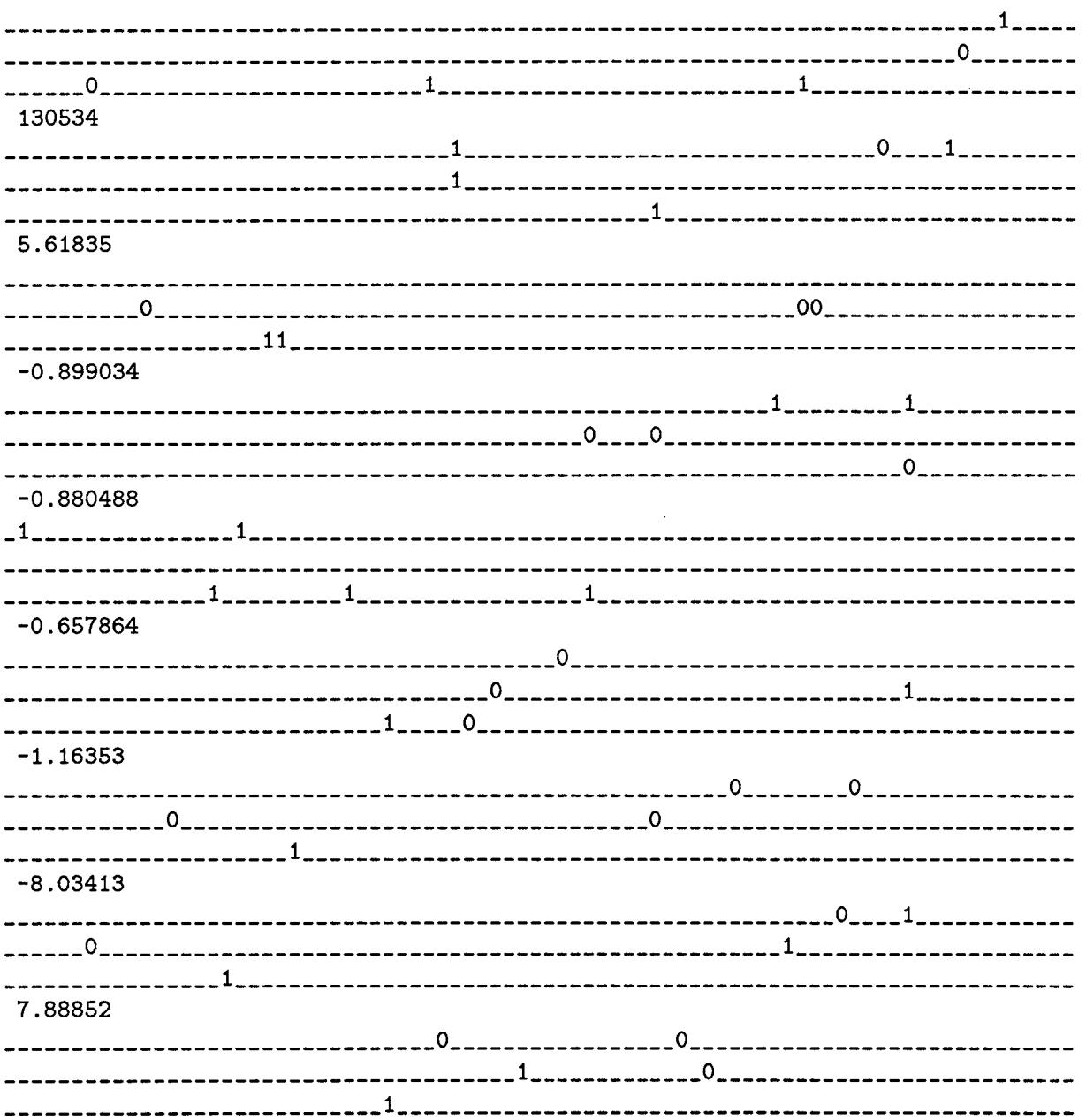
The juxtapositional phase is observed to perform useful processing when it receives a diverse population like that provided by the constant 80% schedule. This schedule generates 64 unique building blocks out of a population of 64.

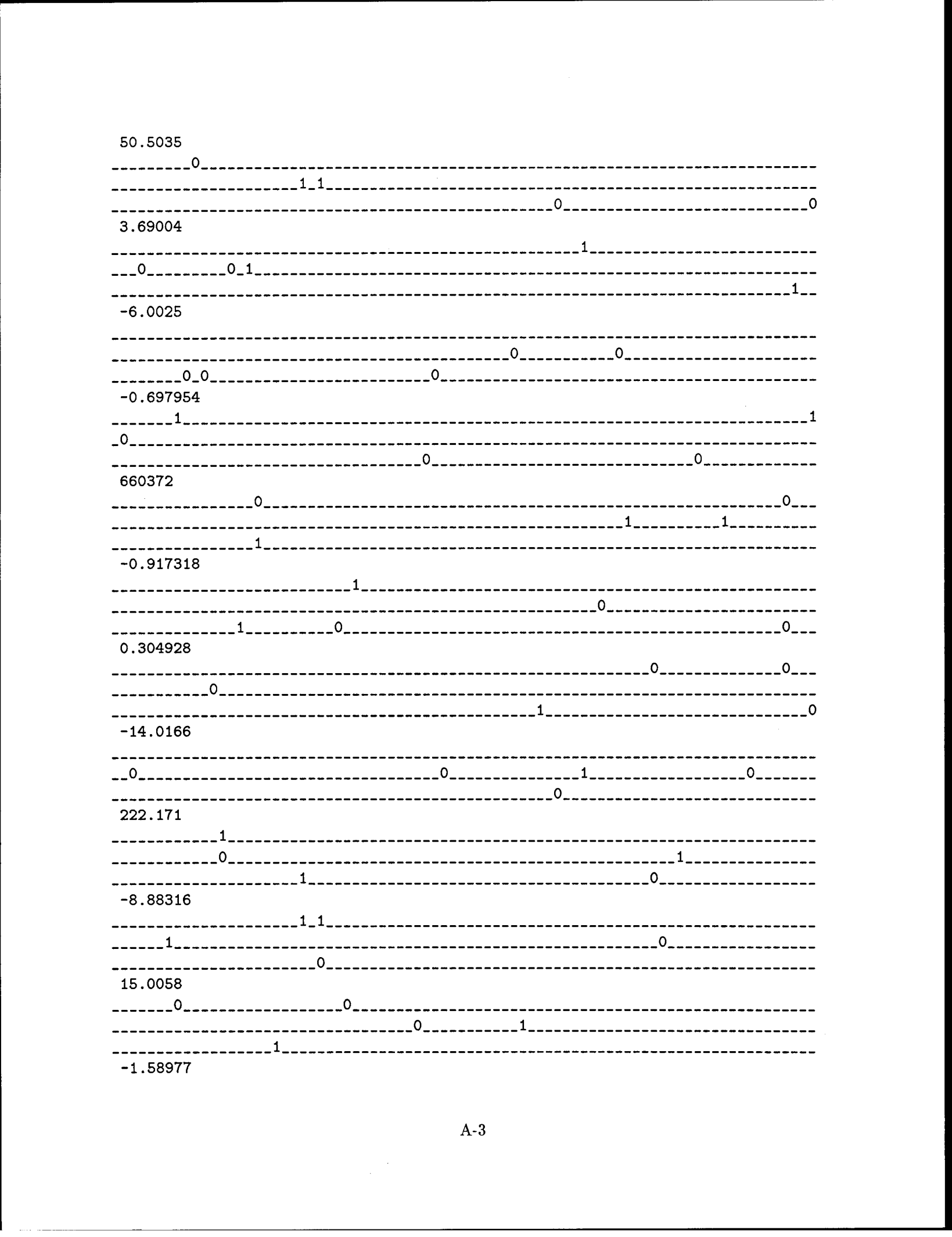
A.2.1 Original Schedule.

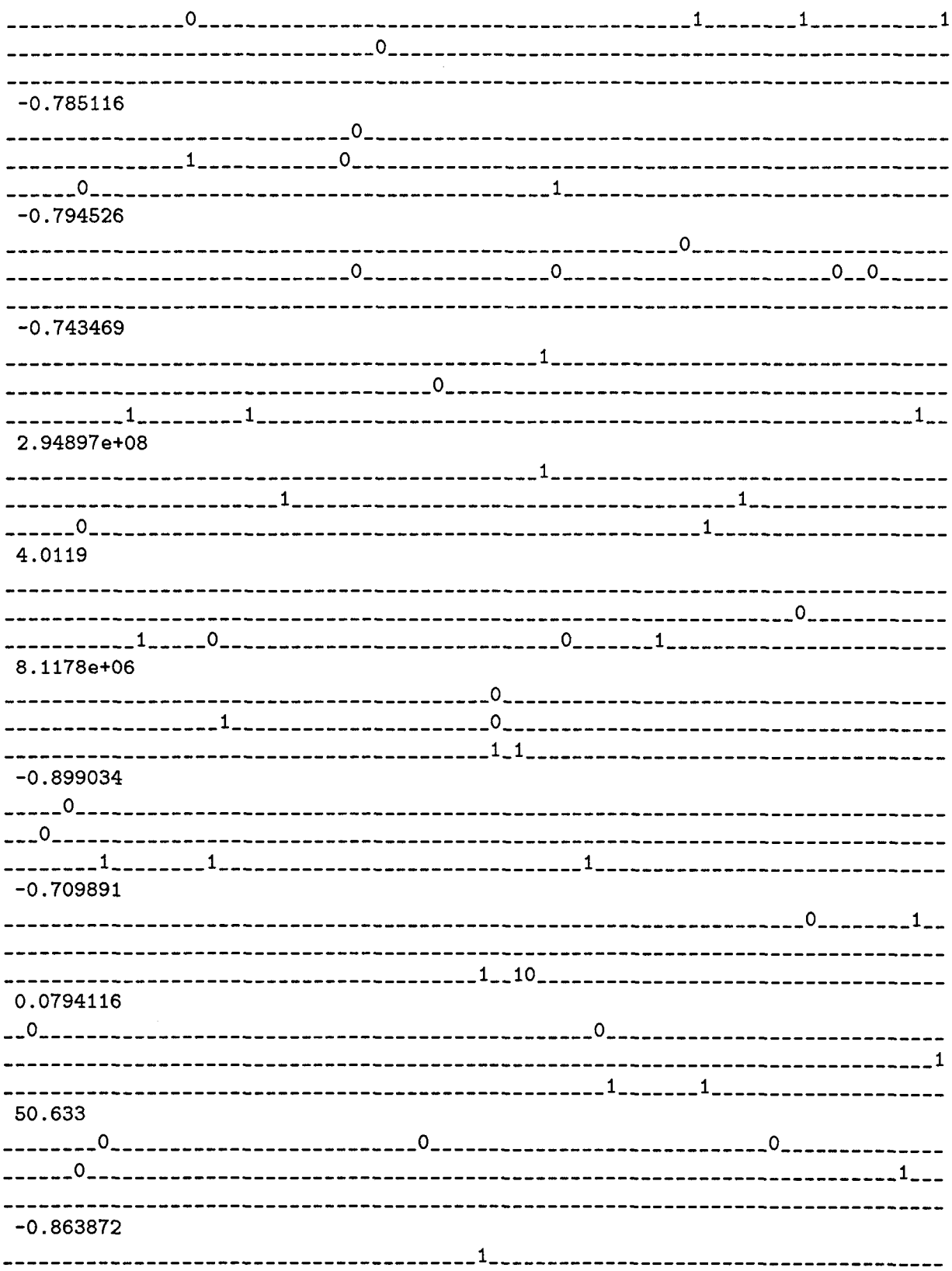
```
-------------------------------------------1-------------------1--------
------------------------0-------------------------------------------------
-------------------------------------------1-------------------0--------
 13.6475
--------------------------------------------------------------------------
```

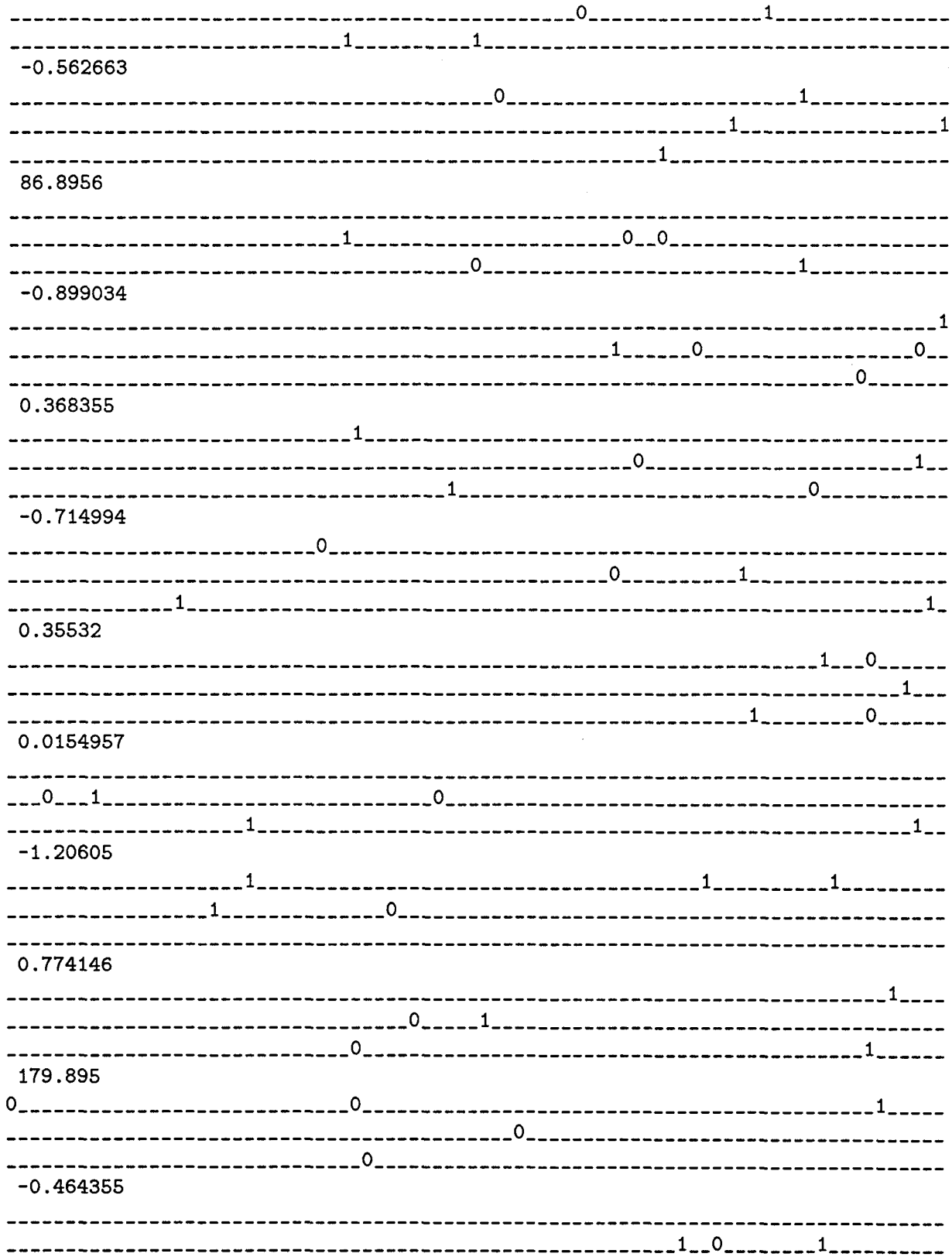


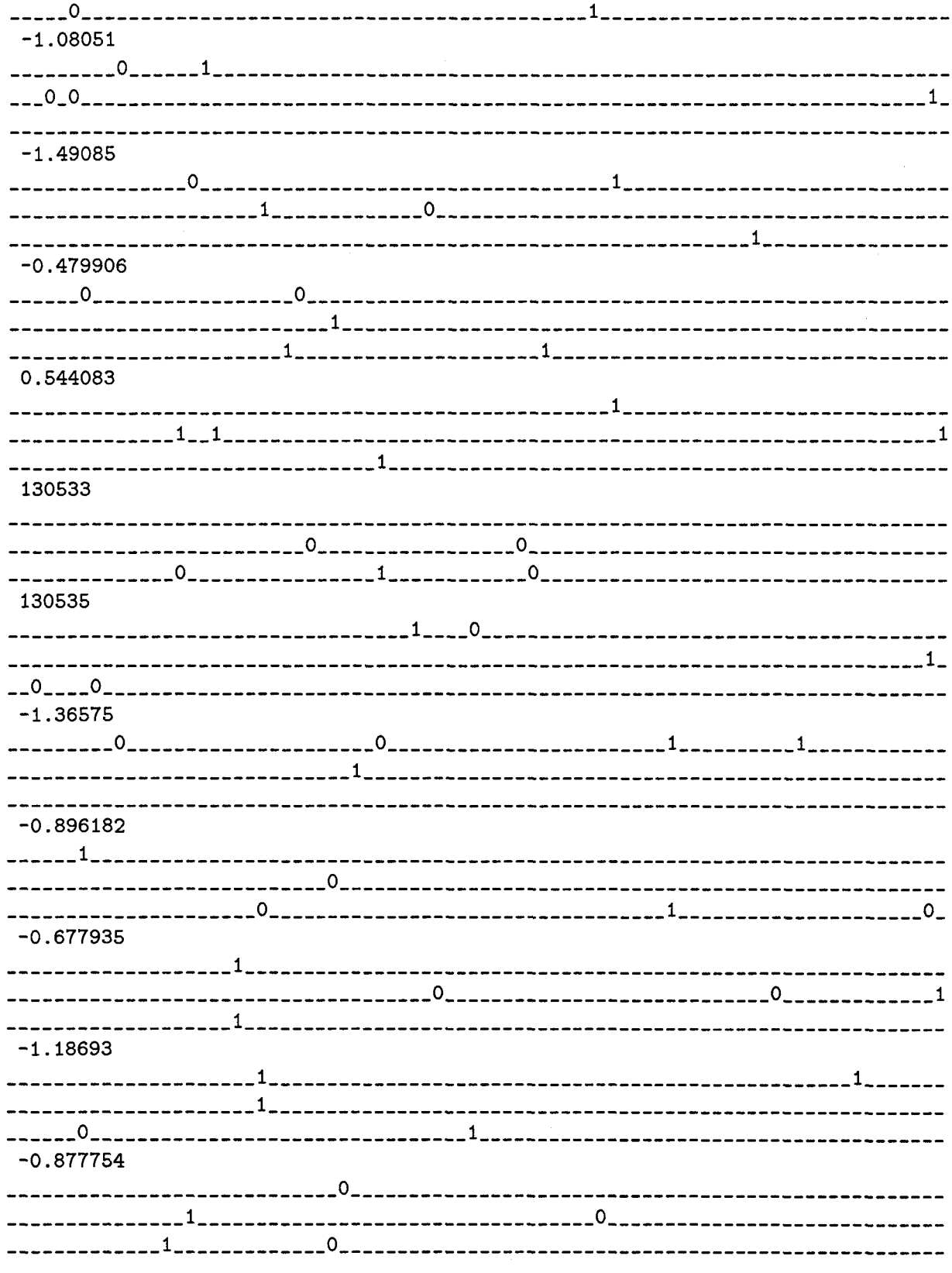
A.2.2 Constant 80% Schedule.

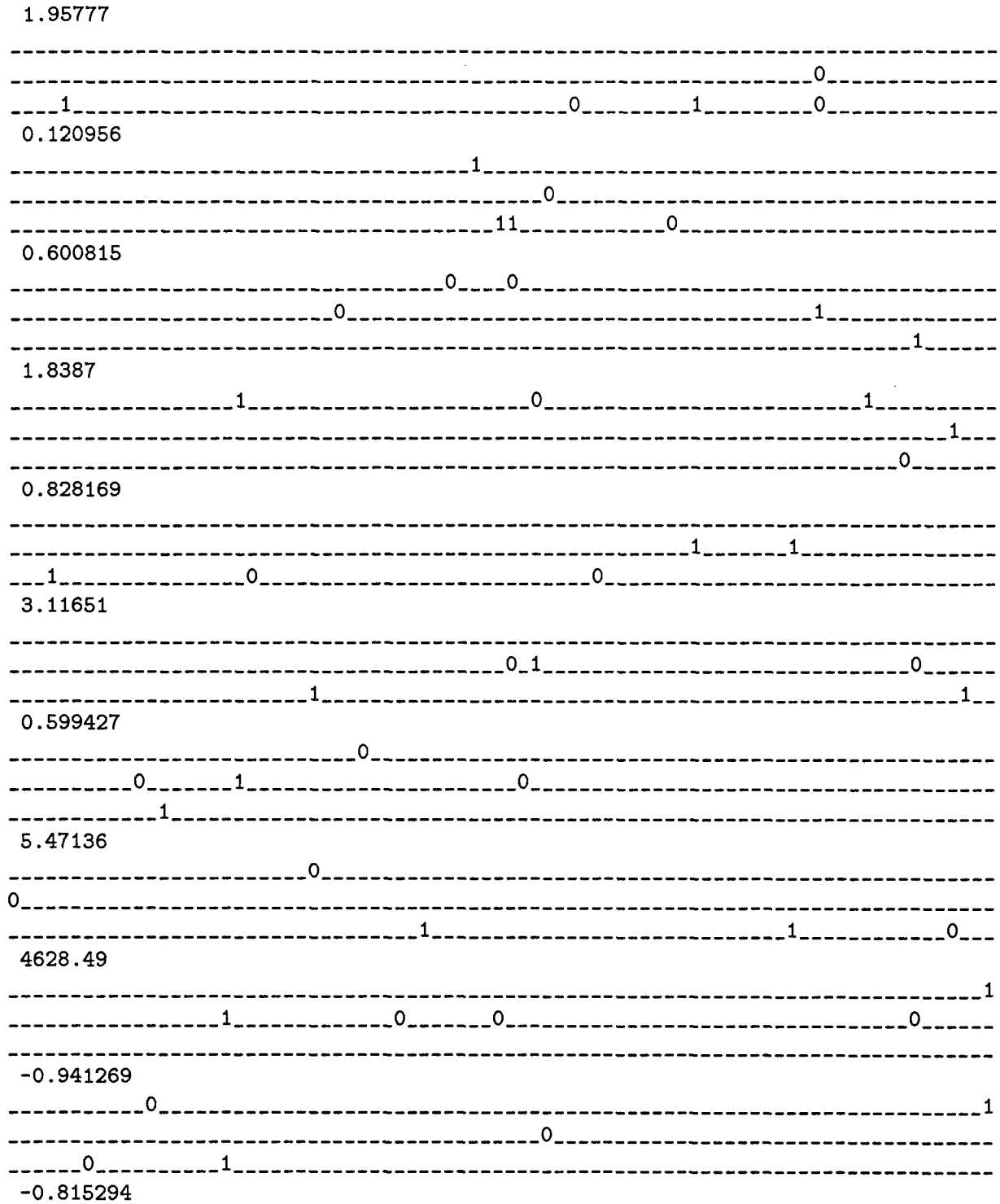



50.5035
3.69004
-6.0025
-0.697954
660372
-0.917318
0.304928
-14.0166
222.171
-8.88316
15.0058
-1.58977


-0.785116
-0.794526
-0.743469
2.94897e+08
4.0119
8.1178e+06
-0.899034
-0.709891
0.0794116
50.633
-0.863872


-0.562663
86.8956
-0.899034
0.368355
-0.714994
0.35532
0.0154957
-1.20605
0.774146
179.895
-0.464355




1.95777
0
1
0
1
0
0.120956
1
0
11
0
0.600815
0
0
0
1
1
1.8387
1
0
1
1
0
0.828169
1
1
1
0
0
3.11651
0
1
0
1
1
0.599427
0
0
1
0
1
5.47136
0
0
1
1
0
4628.49
1
1
0
0
0
-0.941269
0
1
0
0
1
-0.815294

Appendix B. Population Sizing Order-Preserving Transformation

This appendix shows the code and output used to transform the population size calculations to account for tournament selection. See section 5.4.2 for a discussion of the rationale for this transformation.

B.1 Code

```
#include <stdio.h>
#include <math.h>

main()

{
double  fmax  = 5625000000050.0;
double  d     = 0.1;
double  termc = 18048.0;
double  n;
int     i;
extern double log2();

n = termc * fmax * fmax / d / d / 4;
printf("Trans %d: fmax = %e, d = %e, Pop size = %e\n",0,fmax,d,n);
for(i=1;i<101;i++)
   {
   fmax = log2(fmax + 1.0);
   d = log2(d + 1.0);
   n = termc * fmax * fmax / d / d / 4;
   printf("Trans %d: fmax = %e, d = %e, Pop size = %e\n",i,fmax,d,n);
   }
}
```

B.2 Output

```
Trans 0: fmax = 5.625000e+12, d = 1.000000e-01, Pop size = 1.427625e+31
Trans 1: fmax = 4.235499e+01, d = 1.375035e-01, Pop size = 4.281053e+08
Trans 2: fmax = 5.438126e+00, d = 1.858710e-01, Pop size = 3.862285e+06
Trans 3: fmax = 2.686641e+00, d = 2.459471e-01, Pop size = 5.383998e+05
Trans 4: fmax = 1.882307e+00, d = 3.172428e-01, Pop size = 1.588424e+05
Trans 5: fmax = 1.527224e+00, d = 3.975213e-01, Pop size = 6.659685e+04
Trans 6: fmax = 1.337554e+00, d = 4.828703e-01, Pop size = 3.462027e+04
Trans 7: fmax = 1.224999e+00, d = 5.683924e-01, Pop size = 2.095773e+04
Trans 8: fmax = 1.153805e+00, d = 6.492866e-01, Pop size = 1.424823e+04
Trans 9: fmax = 1.106888e+00, d = 7.218421e-01, Pop size = 1.060942e+04
Trans 10: fmax = 1.075113e+00, d = 7.839528e-01, Pop size = 8.485897e+03
```

```
Trans 11: fmax = 1.053190e+00, d = 8.350775e-01, Pop size = 7.176772e+03
Trans 12: fmax = 1.037867e+00, d = 8.758410e-01, Pop size = 6.335811e+03
Trans 13: fmax = 1.027060e+00, d = 9.075375e-01, Pop size = 5.778719e+03
Trans 14: fmax = 1.019389e+00, d = 9.317114e-01, Pop size = 5.401147e+03
Trans 15: fmax = 1.013919e+00, d = 9.498796e-01, Pop size = 5.140889e+03
Trans 16: fmax = 1.010005e+00, d = 9.633850e-01, Pop size = 4.959258e+03
Trans 17: fmax = 1.007199e+00, d = 9.733431e-01, Pop size = 4.831345e+03
Trans 18: fmax = 1.005184e+00, d = 9.806418e-01, Pop size = 4.740666e+03
Trans 19: fmax = 1.003735e+00, d = 9.859680e-01, Pop size = 4.676072e+03
Trans 20: fmax = 1.002691e+00, d = 9.898424e-01, Pop size = 4.629900e+03
Trans 21: fmax = 1.001940e+00, d = 9.926542e-01, Pop size = 4.596812e+03
Trans 22: fmax = 1.001399e+00, d = 9.946913e-01, Pop size = 4.573057e+03
Trans 23: fmax = 1.001009e+00, d = 9.961655e-01, Pop size = 4.555980e+03
Trans 24: fmax = 1.000727e+00, d = 9.972314e-01, Pop size = 4.543692e+03
Trans 25: fmax = 1.000525e+00, d = 9.980015e-01, Pop size = 4.534844e+03
Trans 26: fmax = 1.000378e+00, d = 9.985576e-01, Pop size = 4.528469e+03
Trans 27: fmax = 1.000273e+00, d = 9.989592e-01, Pop size = 4.523875e+03
Trans 28: fmax = 1.000197e+00, d = 9.992490e-01, Pop size = 4.520564e+03
Trans 29: fmax = 1.000142e+00, d = 9.994582e-01, Pop size = 4.518176e+03
Trans 30: fmax = 1.000102e+00, d = 9.996091e-01, Pop size = 4.516455e+03
Trans 31: fmax = 1.000074e+00, d = 9.997180e-01, Pop size = 4.515213e+03
Trans 32: fmax = 1.000053e+00, d = 9.997966e-01, Pop size = 4.514318e+03
Trans 33: fmax = 1.000038e+00, d = 9.998532e-01, Pop size = 4.513672e+03
Trans 34: fmax = 1.000028e+00, d = 9.998941e-01, Pop size = 4.513206e+03
Trans 35: fmax = 1.000020e+00, d = 9.999236e-01, Pop size = 4.512870e+03
Trans 36: fmax = 1.000014e+00, d = 9.999449e-01, Pop size = 4.512627e+03
Trans 37: fmax = 1.000010e+00, d = 9.999603e-01, Pop size = 4.512453e+03
Trans 38: fmax = 1.000008e+00, d = 9.999713e-01, Pop size = 4.512326e+03
Trans 39: fmax = 1.000005e+00, d = 9.999793e-01, Pop size = 4.512235e+03
Trans 40: fmax = 1.000004e+00, d = 9.999851e-01, Pop size = 4.512170e+03
Trans 41: fmax = 1.000003e+00, d = 9.999892e-01, Pop size = 4.512123e+03
Trans 42: fmax = 1.000002e+00, d = 9.999922e-01, Pop size = 4.512088e+03
Trans 43: fmax = 1.000001e+00, d = 9.999944e-01, Pop size = 4.512064e+03
Trans 44: fmax = 1.000001e+00, d = 9.999960e-01, Pop size = 4.512046e+03
Trans 45: fmax = 1.000001e+00, d = 9.999971e-01, Pop size = 4.512033e+03
Trans 46: fmax = 1.000001e+00, d = 9.999979e-01, Pop size = 4.512024e+03
Trans 47: fmax = 1.000000e+00, d = 9.999985e-01, Pop size = 4.512017e+03
Trans 48: fmax = 1.000000e+00, d = 9.999989e-01, Pop size = 4.512012e+03
Trans 49: fmax = 1.000000e+00, d = 9.999992e-01, Pop size = 4.512009e+03
Trans 50: fmax = 1.000000e+00, d = 9.999994e-01, Pop size = 4.512006e+03
Trans 51: fmax = 1.000000e+00, d = 9.999996e-01, Pop size = 4.512005e+03
Trans 52: fmax = 1.000000e+00, d = 9.999997e-01, Pop size = 4.512003e+03
Trans 53: fmax = 1.000000e+00, d = 9.999998e-01, Pop size = 4.512002e+03
Trans 54: fmax = 1.000000e+00, d = 9.999998e-01, Pop size = 4.512002e+03
Trans 55: fmax = 1.000000e+00, d = 9.999999e-01, Pop size = 4.512001e+03
```

```
Trans 56: fmax = 1.000000e+00, d = 9.999999e-01, Pop size = 4.512001e+03
Trans 57: fmax = 1.000000e+00, d = 9.999999e-01, Pop size = 4.512001e+03
Trans 58: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 59: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 60: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 61: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 62: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 63: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 64: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 65: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 66: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 67: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 68: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 69: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 70: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 71: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 72: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 73: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 74: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 75: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 76: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 77: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 78: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 79: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 80: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 81: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 82: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 83: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 84: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 85: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 86: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 87: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 88: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 89: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 90: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 91: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 92: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 93: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 94: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 95: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 96: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 97: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 98: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 99: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
Trans 100: fmax = 1.000000e+00, d = 1.000000e+00, Pop size = 4.512000e+03
```

Appendix C. Protein Visualization and Comparison

This appendix describes the complex process and software tools used to visualize and compare protein conformations. The objective is to document these procedures so that there is a smaller learning curve for follow-on students and researchers. All references made to QUANTA and Cerius in this document are limited to the specific configurations on a Silicon Graphics workstation called Curie at the Materials Laboratory at Wright-Patterson AFB.

C.1 Translation Process for Energy Comparison and Local Minimization

The following process is used to import the PDB file output produced by the genetic algorithms into QUANTA for comparison with other molecules and local energy minimization.

- Execute the command "cerius2" at the command line to run Cerius.
- Select *File→Load Model* to load a molecule.
  - Change the file format to PDB.
  - Load the PDB file that was the output from the genetic algorithm.
- Select *Build→Edit Bonds* to calculate the bonds of the molecule. (NOTE: Check to make sure these bonds are correct. They are produced by distance calculations which may produce a few erroneous bonds.)
- Select *File→Save Model* to save the molecule.
  - Change the file format to PDB.
  - Overwrite the PDB file.
- Exit Cerius.
- Execute the command "quanta" at the command line to run QUANTA.
- Put QUANTA in RTF mode by selecting *CHARMm→CHARMm Mode→RTF.*
- Select *File→Import* to load the PDB file created earlier.
- Select *Edit→Molecular Editor* to edit the bonding properties.
  - Select *Change Bond* and change all the C-O bonds to double bonds.
  - Select *Edit Bonds→Aromatize* and aromatize all the aromatic carbon rings.
- Select *Exit Edit Bonds* and then *Save and Exit.*
  - Ensure an RTF file is generated and the Gusteiger Method of charge distribution is selected when prompted for the save options.

QUANTA can now calculate the CHARMm energy of the molecule. Select *CHARMm Energy* from the Modeling Menu to obtain the energy value. Select *CHARMm Minimization* to locally minimize the energy of the molecule.

C.2 Printing Protein Conformations

Two types of prints were prepared during the course of this work: black and white figures and color prints. The tools used to create these pictures include: QUANTA, snapshot, tops, gammawarp, and lp. To complete either of the visualizations, start QUANTA and load the desired protein. Also, type "snapshot" at the Unix command line and place its icon where it will be accessible.

The process to make encapsulated post script files for inclusion in a Latex document is as follows:

- In QUANTA, change the display to black and white by holding the right mouse button down while selecting *Preferences→Color Definitions→Black and White.*
- Using snapshot, outline the area of the screen to be included in the picture.
- Set a unique image filename in snapshot (usually *filename*.snp).
- Save the image using snapshot.
- Use the command "tops *filename*.snp -eps >*filename*.eps" to translate the image file to encapsulated postscript.

The process to print color pictures is as follows:

- If the current QUANTA display isn't already in its original colors, hold the right mouse button down and select *Preferences→Color Definitions→Reset All.*
- Change the background color to gray:
  - Select *Preferences→Color Definitions→Menu Colors*
  - Select *Background of Viewing Area* and press OK.
  - Set the hue to 0.0, the saturation to 0.2, and the intensity to 0.2 then press OK.
- Using snapshot, outline the area of the screen to be included in the picture.
- Set a unique image filename in snapshot (usually *filename*.snp).
- Save the image using snapshot.
- Use the command "gammawarp *filename*.snp *filename*.gw 0.2" to modify the colors for printing (you can experiment with the number to get the desired color effect).

- Use the command "tops *filename*.snp -RGB >*filename*.rgb" to translate the image file to color postscript.
- Print the *filename*.rgb file using the command "lp *filename*.rgb".

Vita

Captain George H. Gates, Jr. enlisted in the United States Air Force in December of 1982. He was first assigned to the 1916^{th} Communications Squadron at Pease AFB, NH, as a communications-computer systems maintenance technician. Capt Gates was accepted to the Air Force Education and Commissioning Program in 1987, earned his bachelor's degree in Computer Science and Mathematics from Wright State University in 1989, and was commissioned through Officer Training School in April of 1990. He was then assigned to the 50^{th} Space Systems Squadron at Falcon AFB, CO, where he was responsible for the maintenance and administration of all system and application software for the 50^{th} Space Wing Command Post. Capt Gates left Space Command in 1993 to attend AFIT. He has subsequently been assigned to the Electromagnetic Materials Division of Wright Laboratory where he will apply his education to similar research projects.

Permanent address: 108 Dupont Way
WPAFB, OH 45433
(513) 254-8917

Bibliography

1. Bäck, Thomas and others. "Evolutionary Programming and Evolution Strategies: Similarities and Differences." *The Second Annual Conference on Evolutionary Programming*. 11–22. San Diego CA: Evolutionary Programming Society, 1993.
2. Brassard, Gilles and Paul Bratley. *Algorithmics Theory & Practice*. Englewood Cliffs, New Jersey 07632: Prentice Hall, Inc., 1988.
3. Brinkman, Donald J. *Genetic Algorithms and Their Application to the Protein Folding Problem*. MS thesis, AFIT/GCE/ENG/93D-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.
4. Brooks, Bernard R., et al. "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations," *Journal of Computational Chemistry*, *4*(2):187–217 (1983).
5. Cahoon, J. P., et al. "A Multi-population Genetic Algorithm for Solving the K-Partition Problem on Hyper-cubes." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 244–248. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
6. Chan, Hue Sun and Ken A. Dill. "The Protein Folding Problem," *Physics Today*, 24–32 (February 1993).
7. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design*. Reading, MA: Addison-Wesley Publishing Company, August 1988.
8. CHARMm. *CHARMm User's Guide*.
9. Committee on Physical, Mathematical, and Engineering Sciences. *Grand Challenges: High Performance Computing and Communications*. Technical Report, 1800 G Street NW, Washington, D.C. 20550: NSF/CISE, 1991.
10. Committee on Physical, Mathematical, and Engineering Sciences. *Grand Challenges 1993: High Performance Computing and Communications*. Office of Science and Technology Policy, 1992.
11. Davis, Lawrence. "Adapting Operator Probabilities in Genetic Algorithms." *International Conference on Genetic Algorithms*. 61–76. 1989.
12. Davis, Lawrence, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
13. De Jong, Kenneth and William Spears. "On the State of Evolutionary Computation." *Proceedings of the Fifth Interantional Conference on Genetic Algorithms*, edited by Setphanie Forrest. 618–623. San Mateo, CA 94403: Morgan Kaufmann Publishers, Inc., July 1993.
14. De Jong, Kenneth A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD dissertation, University of Michigan, 1975.
15. De Jong, Kenneth A. "Adaptive System Design: A Genetic Approach," *IEEE Transactions on Systems, Man and Cybernetics*, *10*(9) (September 1980).
16. De Jong, Kenneth A. "On Using Genetic Algoriths to Search Program Spaces." *Genetic Algorithms and their Applications: Proceedings of the Second Internation Conference on Genetic Algorithms*. 210–216. Hillsdale, NJ 07642: Lawrence Erlbaum Associates, Publishers, 1987.

17. DeCegama, Angel L. *The Technology of Parallel Processing, Parallel Processing Architectures and VLSI Hardware, Volume I*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1989.

18. Dorigo, Marco and Vittorio Maniezzo. "Parallel Genetic Algorithms: Introduction and Overview of Current Research," *Parallel Genetic Algorithms*, 5–35 (1993).

19. Duncan, Bruce S. "Parallel Evolutionary Programming." *The Second Annual Conference on Evolutionary Programming*. 202–208. San Diego, CA 92121: Evolutionary Programming Society, 1993.

20. Dymek, Andrew. *An Examination of Hypercube Implementations of Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92M-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1992.

21. ECEPP/2. *ECEPP/2*.

22. El-Rewini, Hesham, et al. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall Series in Innovative Technology, Englewood Cliffs, NJ 07632: Prentice Hall, 1994.

23. Eshelman, Larry J., et al. "Biases in the Crossover Landscape." *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. David Schaffer. 10–19. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.

24. Eshelman, Larry J. and J. David Schaffer. "Preventing Premature Convergence in Genetic Algorithms by Preventing Incest." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 115–122. San Mateo, CA: Morgan Kaufman Publishers, 1991.

25. Fogarty, Terence C. "Varying the Probability of Mutation in the Genetic Algorithm." *International Conference on Genetic Algorithms*. 104–109. 1989.

26. Fogel, David B. "Simulated Evolution: A 30-Year Perspective," *IEEE—ACSSC* (1990).

27. Fogel, David B. "On the Philosophical Differences between Evolutionary Algorithms and Genetic Algorithms." *The Second Annual Conference on Evolutionary Programming*. 23–29. San Diego CA: Evolutionary Programming Society, 1993.

28. Fogel, Lawrence J. "The Future of Evolutionary Programming," *IEEE—ACSSC*, 1036–1038 (1990).

29. Forrest, Stephanie and Melanie Mitchell. "Relative Building-Block Fitness and the Building-Block Hypothesis." *Foundations of Genetic Algorithms 2*. Morgan Kaufmann Publishers, Inc., 1993.

30. Garey, Michael R. and David S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. San Francisco, CA: W. H. Freeman and Company, 1979.

31. Gates, Jr., George H. "Combinatoric Algorithm (NP-Complete) Design Project, The Protein Folding Problem." CSCE 686 Advanced Algorithm Design, June 1994.

32. Gates, Jr., George H., "Raw Data in Support of Thesis." In /usr/genetic/Data/ghg.dat directory on Thor, December 1994.

33. Goldberg, David E. *Optimal Initial Population Size for Binary-coded Genetic Algorithms*. Technical Report TCGA Report Number 850001, University of Alabama, Alabama 35486: The Clearing House for Genetic Algorithms, Department of Engineering Mechanics, November 1985.

34. Goldberg, David E. *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1989. Reprinted with corrections.

35. Goldberg, David E., et al. *Genetic Algorithms, Noise, and the Sizing of Populations*, chapter 6, 333–362. Complex Systems Publications, Inc., 1992.

36. Goldberg, David E., et al. "Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms." *Proceedings of the Fifth International Conference on Genetic Algorithms*, edited by Stephanie Forrest. 56–64. San Mateo, CA: Morgan Kaufmann Publishers, July 1993.

37. Goldberg, David E., et al. "Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale." *Complex Systems*. 415–444. 1990.

38. Goldberg, David E., et al. "Don't Worry, Be Messy." *International Conference on Genetic Algorithms*. 24–30. 1991.

39. Goldberg, David E., et al. "Messy Genetic Algorithms: Motivation, Analysis, and First Results." *Complex Systems*. 493–530. 1989.

40. Goldberg, David E. and Jon Richardson. "Genetic Algorithms with Sharing for Multimodal Function Optimization." *Proceedings of the Second International Conference on Genetic Algorithms*. 41–49. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.

41. Gordon, V. Scott and Darrell Whitley. "Serial and Parallel Genetic Algorithms as Function Optimizers." *Proceedings of the Fifth International Conference on Genetic Algorithms*, edited by Stephanie Forrest. 177–183. San Mateo, CA: Morgan Kaufmann Publishers, Inc., July 1993.

42. Grefenstette, J. J. "Optimization of Control Parameters for Genetic Algorithms," *IEEE Transactions on Systems, Man & Cybernetics*, 122–128 (1986).

43. Grefenstette, John J. "Learning by Analogy in Genetic Classifier Systems." *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. David Schaffer. 291–297. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.

44. Grefenstette, John J. *A User's Guide to Genesis 5.0*. Technical Report, Nashville, TN: Vanderbuilt University, 1990.

45. Grefenstette, John J. "Lamarckian Learning in Multi-agent Environments." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 303–310. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.

46. Grefenstette, John J. *Deception Considered Harmful*. Foundations of Genetic Algorithms 2, Morgan Kaufmann, 1992.

47. Hoare, C. A. R. *Communicating Sequential Processes*. London: Prentice-Hall International, 1984.

48. Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press, 1975.

49. Holland, John H. "Genetic Algorithms," *Scientific American*, *267*(1):66–72 (July 1992).

50. Intel. *iPSC/860 Basic Math Library User's Guide*, April 1991.

51. Jaenicke, R. "Protein Folding: Local Structures, Domains, Subunits, and Assemblies," *Biochemistry*, *30*:3147–3161 (1991).

52. Kronsjö, Lydia and Dean Shumsheruddin, editors. *Advances in Parallel Algorithms*. New York: Halsted Press, 1992.

53. Kuck & Associates. *CLASSPACK Basic Math Library/C User's Guide* (Release 1.3 Edition). Champaign, IL 61820, November 1993.

54. Kumar, Vipin, et al. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

55. Lamont, Gary B., et al. "Evolutionary Algorithms." Compendium of Parallel Programs for the Intel iPSC Computers, Volume V.

56. Larson, Roland E. and Robert P. Hostetler. *Calculus with Analytic Geometry* (3rd Edition). Lexington, MA: D.C. Heath and Company, 1986.

57. LeGrand, Scott M. and Kenneth M. Merz Jr. "The Application of the Genetic Algorithm to the Minimization of Potential Energy Functions," *Journal of Global Optimization*, 49–66 (1993).

58. Lengauer, Thomas. "Algorithmic Research Problems in Molecular Bioinformatics," *Arbeitspapiere der GMD 748* (May 1993).

59. Levi, Shem-Tov and Ashok K Agrwala. *Real Time System Design*. McGraw-Hill Computer Science Series, New York: McGraw-Hill Publishing Company, 1990.

60. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall, 1992.

61. Martin, IV, Robert C. *A Gain Scheduling Optimization Method using Genetic Algorithms*. MS thesis, AFIT/GAE/ENG/94D-, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1994.

62. Merkle, Laurence D. *Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

63. Merkle, Laurence D. Personal Conversation, October 1994.

64. Merkle, Laurence D. and George H. Gates, Jr., "Paragon Implementation of a Parallel Fast Messy Genetic Algorithm." In "/usr/genetic/Toolkit/Messy/ParFast" directory on Thor, June 1994.

65. Merkle, Laurence D., Gates Jr., George H., Lamont, Gary B., and Pachter, Ruth. "Application of the Parallel Fast Messy Genetic Algorithm to the Protein Folding Problem." *Proceedings of the Intel Supercomputer Users Group 1994 Annual North America Users Conference*, edited by JoAnne Wold. 189–195. June 1994.

66. Michalewicz, Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.

67. Mühlenbein, H., et al. "The Parallel Genetic Algorithm as Fucntion Optimizer," *Paralle Computing*, *17*(7):619–632 (1991).

68. Nayeem, Akbar and others. "A comparative Study of the Simulated-Annealing and Monte Carlo-with-Minimization Approaches to the Minimum-Energy Structures of Polypeptides: [Met]-Enkephalin," *Journal of Computational Chemistry*, *12*(5):594–605 (1991).

69. Office of Technology Assessment. *Mapping Our Genes—The Genome Projects: How Big, How Fast?*. Technical Report No. OTA-BA-373, U. S. Government Printing Office, Washington, D.C.: U. S. Congress, 1988.

70. Olsan, James B. *Genetic Algorithms Applied to a Mission Routing Problem*. MS Thesis, AFIT/GCE/ENG/93-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

71. Pachter, Ruth, et al. "Smart Structures and Materials," *SPIE Proceedings 1* (1993).

72. Parish, Donald A. *A Genetic Algorithm Approach to Automating Satellite Range Scheduling*. MS thesis, AFIT/GOR/ENS/94M-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1993.

73. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley Publishing Company, 1984.

74. Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

75. Pettey, Chrisila C. and Michael R. Leuze. "A Theoretical Investigation of a Parallel Genetic Algorithm." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.

76. Sawyer, George A., et al., "Hypercube Implementation of a Parallel Simple Genetic Algorithm." In "/usr/genetic/Toolkit/PSGA" directory on Thor, April 1994.

77. Schaffer, J. David, et al. "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization." *Third International Conference on Genetic Algorithms*. 51–60. 1989.

78. Schaffer, J. David and Larry J. Eshelman. "On Crossover as an Evolutionarily Viable Strategy." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 61–68. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.

79. Spears, William M. and Kenneth A. De Jong. "Using Genetic Algorithms for Supervised Concept Learning," *IEEE-CH*, *29*(15):335–341 (July 1990).

80. Spiessens, Piet and Bernard Manderick. "A Massively Parallel Genetic Algorithm: Implementation and First Results." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 279–285. San Mateo, CA: Morgan Kaufmann Publishers, 1991.

81. Srinivas, M. and Lalit M. Patnaik. "Genetic Algorithms: A Survey," *COMPUTER*, *27*(6):17–26 (June 1994).

82. Starkweather, T., et al. "A Comparison of Genetic Sequencing Operators." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 69–76. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.

83. Sterling, Thomas, et al. *Enabling Technologies for Peta(FL)OPS Computing*. Technical Report CCSF-45, California Institute of Technology, July 1994.

84. Tanese, Reiko. "Parallel Genetic Algorithms for a Hypercube." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, edited by John J. Grefenstette. 177–183. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, July 1987.

85. von Freyberg, Berthold and Werner Braun. "Efficient Search for All Low Energy Conformations of Polypeptides by Monte Carlo Methods," *Journal of Computational Chemistry*, *12*(9):1065–1076 (1991).

86. Walpole, Ronald E. and Raymond H. Myers. *Probability and Statistics for Engineers and Scientists* (Third Edition). 866 Third Avenue, New York, NY 10022: MacMillan Publishing Company, 1985.

87. Whitley, Darrel. "The *GENITOR* Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best." *International Conference on Genetic Algorithms*. 1989.

88. Whitley, Darrell, et al. "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 133–140. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	13 Dec 94	Master's Thesis

4. TITLE AND SUBTITLE
Predicting Protein Structure Using Parallel Genetic Algorithms

5. FUNDING NUMBERS

6. AUTHOR(S)
George H. Gates, Jr., Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Wright Laboratory (AFMC)
Materials Directorate
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The protein folding problem is a biochemistry Grand Challenge problem. The challenge is to reliably predict natural three-dimensional structures of polypeptides. Genetic algorithms (GAs) are robust, semi-optimal search techniques modeling natural evolutionary processes. Fast messy GAs (fmGAs) are variants of messy GAs that reduce the exponential time complexity to polynomial. This investigation evaluates the merits of parallel SGAs and fmGAs for minimizing the potential energy of a pentapeptide, [Met]-enkephalin.

AFIT's energy model is compared to a similar model in a commercial package called QUANTA. Differences between the two models are identified and resolved to enhance GAs' abilities to correctly fold molecules. The steps required to unify the behavior of the two implementations is presented.

The effectiveness of SGAs while minimizing the potential energy of [Met]-enkephalin is shown to be highly dependent on the choice of population size and mutation rate. It is also demonstrated that choosing parameters from the Schaffer's proposed guidelines cause SGAs to realize near-optimal performance on this particular application.

Parallel SGAs are capable of finding near-optimal conformations of [Met]-enkephalin. Parallel fmGAS should ultimately find better solutions in less time. The experiments performed in this investigation determine limitations of parallel SGAs and fmGAs applied to polypeptide energy minimization.

14. SUBJECT TERMS
ParallelProcessing, Genetic Algorithms, Protein Folding Problem

15. NUMBER OF PAGES
120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL